Department of Computer Science

Technical Report

Undergraduate Dissertation: TOADS: A tool to aid in the Development of MAS consisting of OCLP-minded agents, under JADE

Tadeo M. Corradi

Technical Report 2004-16 **May 2004**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

TOADs: A tool to aid in the development of MAS consisting of OCLP-minded agents, under JADE.

Tadeo M. Corradi

BsC in Mathematical Sciences

May 2004

TOADs: A tool to aid in the development of MAS consisting of OCLP-minded agents, under JADE.

Submitted by: Tadeo M. Corradi

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:.....

Date:.....

Abstract

Tadeo M. Corradi
Department of Computer Science, University of Bath at Bath, BA2 7AY, United Kingdom

Logic Programming (LP) is a growing field in the agents programming community. When faced with the task of representing a problem with logic rules, it is crucial that the formalism used allows for an intuitive way of expressing the problem: OCLP provides a direct representation of choices and preference. Following the work of [De Vos 2001] in the creation of OCLP, we analyse how this formalism can be used and implemented to represent reasoning capabilities in an Intelligent Agent System. In order to aid in the development of such multi-agent system (MAS), a bridge is needed between an answer set solver and an agent platform. Motivated by this, TOADs (Tractable OCLP Agent Definitions) is designed and developed. TOADs is intended to be a framework for the creation of OCLP-minded agents and multi-agents systems. The scope of the project is extended and a GUI is provided to help supervise and edit OCLP agents as they interact allowing the user to follow the system as it evolves.

Acknowledgements

I would like to manifest my gratitude to the following people:

My project supervisor, Dr. Marina de Vos, whose help and guidance were crucial for the writing of this dissertation.

My mother, without whose love and wisdom I could have never started this project in the first place.

My dear friends Federico Ceccatto and Facundo Ercole, whose support remained unconditional beyond distance and circumstances.

To my dear grandmother “La Yaya”.

Contents

Chapter 1

Introduction

Logic Programming, as opposed to procedural programming, gives a direct way of expressing both data and relations between pieces of data. The main drawback of Logic Programming when used to define the reasoning capabilities of an Intelligent Agent is, however, its lack of expressive power. Often, a logic program requires too many rules to represent a relation which is simple when expressed in natural language. Sentences like *I ought to choose between moving the king, the queen or the rook* or ideas like *A decision involving my survival is more important than a decision about this game of chess* are very difficult, if at all possible, to express.

The Ordered Choice Logic Programming formalism's syntax can express these two kinds of notions (choice and preference). This means that an Intelligent Agent whose reasoning capabilities are defined by an OCLP program can be easily built and maintained.

This is of particular importance in agent programming, allowing for an agent's goals and sub-goals to be explicitly defined by preference relations. Furthermore by interchanging information a simple way of argumentation can be simulated, whereby the agents review their beliefs based on other agents' beliefs and aim to reach a general consensus. Negotiating agents may benefit from this, for example if several departments in a University need to decide upon the room allocation; supposing each department is represented by an agent, in this case choice logic would allow for notions like *a certain room cannot be used by two lecturers at the same time* and preference, ideas like *the Computer Science Department prefers to have the University Hall at their disposal on Wednesdays rather than Fridays.*

However, the resulting framework is quite complex and difficult to manage from a programming point of view; therefore a tool is needed to help in both the development and supervision of such multi-agent systems. TOADs (Tractable OCLP Agent Definitions) is designed with these requirements in mind and it will provide a suitable data structure, the JADE agent definitions for such agents, a GUI to supervise a running MAS and several services to reuse this architecture, including an interface with the answer ser solver OCT.

As explained by [De Vos 2001], agents specified in this way can interact with each other by sharing their beliefs (the answer sets of their OCLP program) and updating their program upon this incoming information, so as to increase their knowledge about the world. Systems of this kind can have very interesting properties (for one, they represent the evolution of knowledge bases till a general consensus is achieved) including, as stated by [De Vos 2001] and exemplified in chapter 7, the fact that one can map extensive games to a system of this kind and when the system becomes stable, the Answer Sets for each agent correspond to the Nash Equilibria of the game.

To aid in the development of these OCLPAS (OCLP Agent Systems), a tool is needed. Fundamentally, an interface is needed between some agent development framework and an answer set solver; but also an ontology is needed for the grounding of notions like "rule" or "OCLP program" and also a protocol for the interaction between OCLP agents. The second and main aim of this project is to develop these definitions, packaged in an application named TOADs. As a part of this package we intend to include a simple GUI agent, and the option of running OCLP agents in a retained mode (waiting for orders before doing anything) awaiting orders from the user (through the GUI agent). The motivation for this is that distributed systems are usually difficult to supervise or debug, even more so if they include declarative programs.

Logic Programming, OCLP, multi-agents systems, an MAS platform and an OCLP answer set solver are introduced in Chapter 2.

In Chapters 4 and 5 the development of TOADs is presented. Chapter 6 illustrates the use of TOADs.

We conclude with the test of the final product, including related work and enhancement proposals.

Chapter 2

Literature review

2.1 Description of this chapter

The literature review is organized as follows:
In Section 2 multi-agent systems are introduced, their general characteristics, among them beliefs, as a motivation for the use of logic programming.
In Section 3, several logic programming formalisms are mentioned along with some applications in multi-agent systems.
Section 4 gives a detailed description of OCLP, its application to MAS, related work and examples.
In section 5, JADE is discussed, providing a general overview and the relevant features for this project are analysed in more detail, identifying their utility in solving the needs of a multi-agent system composed of OCLP agents. Section 6 quickly mentions OCT and OCT input structure.

2.2 Multi-agent systems

A multi-agent system (MAS) is a collection of interacting organisms in an environment. Viewing a system as a multi-agent system is a matter of organisational perspective: they are a consequence of autonomy assumption and interaction. These are not strong assumptions, so, for example: specialization and task distribution, free market or a football match can be viewed as multi-agent systems. In any of those we are in the presence of autonomous entities pursuing a goal, in collaboration, competition, or any other level of interaction. By running a virtual MAS we can analyze the emerging properties of the system overall, both with the purpose of simulating their real counterparts and designing distributed solutions where centralized applications are not appropriate, excessively complicated or just not practical.

For example, if one wants to create a program that acts as the artificial intelligence for a game of virtual football, a classical centralised solution would be to program the manager's mind and where players would be dummy terminals which

carry out orders. A program of this shape would have to process variables for twelve terminals and the algorithm would tend to be intricate and probably unintelligible. A distributed MAS solution would be to program each player as an agent, with a collection of common behaviours (for example everyone would have a doNotScore-OwnGoal behaviour, but a defense will have a keepCloseToAnAdversary behaviour while a forward will have a findAClearSpot behaviour). Under this organization, the program is easier to understand for a human being, hence easier to analyse, debug and adapt.

However, according to [Wooldridge 2002], in the field of computer science, distributed platforms of autonomous interacting processes (agents) were only introduced very recently (in the 1980's) and gained more attention only after 1990.

The idea behind agents is that of creating a new upper level of abstraction to reason about a computing machine, often catalogued on top of the object abstraction. The motivation for this is the need to model complex behaviour and to introduce characteristics for objects which describe this complexity in a more abstract level, allowing humans to reason and communicate with each other ideas about it.

Among these new characteristics we highlight: *autonomy, intention, activity, intelligence.*

Autonomy is the only property accepted in every context, it states that an agent performs whatever operation on her own and only interacts with other processes by data interchange; a system where two components effectively share the process control should therefore be modeled as a single agent.

Intention or *intentional instance* is an abstraction tool which enables us to talk about an agent's goals. This is a major philosophical step as it liberates computer programs from human tyranny, allowing agents to reject a request. It is possible to have two or more programs which balance their actions to favour each others' goals, but again these should be modelled as a single agent.

Activity reflects the capability of an agent to begin an interaction of some sort; in human terms could be called initiative; depending on what triggers the action, we have pro-active, re-active and hybrid behaviours. While being a feature of many agents, purely reactive agents who merely communicate are perfectly valid.

Intelligence is a delicate concept to discuss. Whichever kind of intelligence (emotional, religious, rational...) is to be modelled, an agent is supposed to follow its rules towards the achievement of the agent's goals. Of particular interest for this project are rational agents, which follow logical methodological deductive rules formalised in several ways: functional programming, procedural programming, declarative programming. We focus on the latter and its logic programming instance.

A general definition of agent

Agents can be defined as a set of states **S**, possible actions **A**, perceptive images or **percepts** (**Per**) and an inner function **f**: PxS $\rightarrow$ AxS which maps the pair (percept, current state) to the pair (action to perform, new state). This definition is rather

general: components are very flexible ideas, no significant is given and in particular the function f sees no constraints.

2.2.1 The environment

A generally accepted assumption is that percepts depend on some environmental state, which changes over time. This change is forced to be modelled as discrete, which is not a problem since any continuous environment can be modeled as discrete with arbitrary level of accuracy [Wooldridge 2002]. Furthermore it is sometimes interesting to analyze systems where the environment changes deterministically allowing to talk about an environment state update function.

2.2.2 Actions

The means by which an agent expects to achieve his goals, actions are again a fairly general idea. Therefore, in general terms, an action can be defined as the process which modifies the environment in some way; an action thus potentially modifies the way the agent or other agents perceive the environment.

In the hope of identifying similarities with human beings and society, it is often clarifying to distinguish between physical actions (e.g. a robot moving around) and communication (e.g. a robot sending some signal). However, these are wrapped together by Speech Act Theory [Searl 1969]. This theory sustains that elocution (the act of speaking) is an intentional act (an action), proposing that intentions must be known in order to capture the fundamental information of a message. Speech act theory has been well accepted and used in agent programming since it is consistent with the intentional view of agents and since it prescribes a general classification for messages (according to intention); this classification is being used by some implementations of communication protocols including the one studied in this project (FIPA ACL Communication) [Vidal 2003].

2.2.3 Goals

Following the general architecture given above, goals would be encoded in the function f, since the agent's conditioning instinct is that of believing his goals will be achieved by following the rules which define f. Pro-active agents are programmed with explicitly determined goals. A standard way of formalising goals is by means of a utility function over the environment, matching every known state to some real number, called utility; conventionally higher utility meaning preferred states and the highest possible utility determines the goal. The utility function is thus defined as

$$u: \text{Per} \rightarrow \text{R}$$

A goal g is a member of Per such that

$$u(g) = \max(\mathrm{Im}(u))$$

where Im(u) stands for the image of the map u.

2.2.4 Beliefs

In the above architecture a set S of states is mentioned, this is sufficient for theoretical purposes as it is an equivalent structure to any other computational representation of knowledge [Turing 1936], but highly un-intuitive when modeling an intelligent agent. One needs to subdivide the agent into various parts which may adopt several states of their own, independently from each other.

A more complex structure is adopted in which the agent holds a set of beliefs emphB, containing a finite number of labels, each one of which acquiring a particular state at every point in time. For this project and generally in Logic Programming, we restrict this idea and name labels to literals which can be in one of three states: true, false or unknown.

In the next section, we to introduce several Logic Programming formalisms as ways of formalising how an agent represents and reasons about beliefs.

2.3 Logic Programming

The main idea of Logic Programming (LP) is [Malpas 1987] to give a paradigm for computation, in logic programming computation is logical deduction.

Contrary to procedural computation, where processes are described explicitly, in logical programming what is given is the description of some objects, the rules that relate them and some constraints to be satisfied.

It is up to the computer to find a model which is consistent with all of these. In LP, these are sentences of one of the following forms:

rule	an English interpretation
$a \longleftarrow$	a is true
$\longleftarrow b$	b is false
$c \longleftarrow a,\ b$	c is true if both a is true and b is true

Where a, b, c are atoms (irreducible expressions which can either be true or false). Generally this set of rules is called a **Logic Program**, and the possible models (sets of true and false atoms consistent with the program rules) are the answer sets. Thus, having a logic program, we can state an arbitrary expression or expressions and the system will be able to tell us when they are either

- consistent with some (true)
- inconsistent with all (false) answer sets

EXAMPLE 1. The truth about spaghetti. *Consider the program:*

rules	an English interpretation
food ⟵ *pasta*	pasta is food
pasta ⟵ *italian*, *quick*	quick italian things are pasta
quick ⟵ *pasta*	pasta is quick
italian ⟵ *pasta*	pasta is italian
quick ⟵ *spaghetti*	spaghetti is quick (to prepare)
italian ⟵ *spaghetti*	spaghetti is italian

The possible models for this program are: {*spaghetti, italian, quick, pasta, food*}; {¬*spaghetti, italian,* ¬*quick,* ¬*pasta, food*}; {¬*spaghetti,* ¬*italian, quick,* ¬*pasta, food*}; {¬*spaghetti,* ¬*italian, quick,* ¬*pasta,* ¬*food*}; {¬*spaghetti, italian,* ¬*quick,* ¬*pasta,* ¬*food*}.

Hence running the logic program with the statement "*food* if *spaghetti*" (interpreted maybe as "spaghetti is food") will return "true". While running it with "¬*pasta* if *spaghetti*" will return "false".

Statements such as "food if spaghetti" in the previous example are called **Theorems**, the rules in the program are called **Axioms** and the "running" procedure is called **Theorem proving**.
A set of assumptions of the values of atoms is called an **Interpretation**. For example: I = {pasta} is an interpretation. Formal definitions for these are given in Section 4.

It is often useful to represent the computation as a function of the triple (Program, Interpretation, Theorem), but effectively this is equivalent to the pair (Program ⋃ "Interpretation as assertion", Theorem).

When a program is run with the intention of obtaining all possible models for a given interpretation, the outputs of this program are called answer sets. An extension from this basic model adds quantifiers to the syntax, allowing rules of the form:

"For all *pasta*, *italian*." Or "There exists a *food*, *quick*". [Richardson 2003] gives a series of algorithms to translate any first-order logic statement into clausal form, a disjunctive list of statements of the following form:

conclusion if (*condition*$_1$ and *condition*$_2$ and...and *condition*$_n$)

A popular logic programming software implementing this syntax is PROLOG. An introduction to PROLOG can be found in [Malpas 1987].

2.3.1 On other Logic formalisms

So why not just first-order logic?

In attempting to model agents there are some abstract notions we ought to define explicitly. For example, whatever our approach is, beliefs and intentions are to be represented in our logic. Thus, as described in [Hoek & Wooldridge 2003], we need sentences of the form

i - Believes - Φ
i - Wants - Φ

This, in first-order logic, could only be expressed as predicates such as Believes(i, Φ) and Wants(i, Φ). This itself is not allowed in first-order logic, since Φ is a sentence, not a term. Therefore we need to extend our logic and produce a new formalism to allow this kind of statements to be expressed and handled.
In the following subsection, a description of the most relevant formalisms for agent programming is given.

Cohen and Levesque's Intention Logic

The original idea in Cohen and Levesque's work [Cohen & Levesque 1990] was to create a logic formalism to allow for manipulation of sentences expressing intention (as in "I intend to..."). Following the requirements of an intentional agent, Cohen and Levesque identify the following atomic expressions for their theory of intention:

Operator	Meaning
(Bel i Φ)	agent i believes Φ
(Goal i Φ)	agent i has goal Φ
(Happens α)	action α will happen next
(Done α)	action α has just happened

Table 1. Atomic modalities in Cohen and Levesque's logic.

Briefly, this formalisms adopts a [Huth & Ryan 2000] KD45 modal logic for beliefs and a KD for goals, where goals are a subset of beliefs (i.e. an agent does not intend to achieve something he believes impossible to). KD45 and KD are explained in detail in [Huth & Ryan 2000].

Furthermore, this formalism adds temporal operators (Happens and Done) to allow the system to reason about the future and whether or not it is possible to achieve a certain goal. In developing their theory of intention, Cohen and Levesque were trying to satisfy Bratman's criteria [Bratman 1987] and [Bratman 1990], which they summarize as follows:

1. Intentions pose problems for agents, who need to determine ways of achieving them.

2. Intentions provide a filter for adopting other intentions, which must not conflict.

3. Agents track the success of their intentions, and are inclined to try again if their attempts fail.

4. Agents believe their intentions are possible.

5. Agents do not believe they will not bring about their intentions.

6. Under certain circumstances, agents believe they will bring about their intentions.

7. Agents need not intend all the expected side effects of their intentions.

BDI logics

Belief-Desire-Intention (or BDI) logics distinguishes three main notions:
Beliefs: the knowledge that an agent has about the universe, called belief since it can be wrong.
Desires: basically a set of feasible sets of beliefs which the agent would like to be true.
Intentions: those desires which the agent allocates resources to achieving by making a commitment.
The motivation for this formalism is the homonym theory by Bratman [Bratman 1987], in which intentions are justified as tools for optimising the decision making procedure: by having an intention -or, equivalently, a commitment towards a desire- the agent can immediately discard any course of action which would render this intention unobtainable.

The most popular implementation of BDI is Procedural Reasoning System (PRS) [Georgeff & Lansky 1987].
While highly successful, this formalism escapes from our interests, since we attempt to model agents which have no intentions beyond increasing their knowledge about the universe.

Fuzzy Logics and Ambiguity

Motivated by the goal of modelling human conclusions drawn from imprecise (vague) information, alternative ideas originated even back in ancient Greece, in particular, as a reaction to Parmenides' "Law of the Excluded Middle", which mandates that every statement has to be either true or false. Heraclites rejected this principle, he suggested that a statement could be simultaneously true and false; this line of thought abstracts above logics assumptions, into philosophy and beyond the scope of this project.

However these counter-arguments against the efforts of Aristotle and philosophers that preceded him led more recent scientist to suggest, propose and finally formalize a multi-valued logic and fuzzy logic, the final step. In fuzzy logic, a statement has a degree of truth represented by a real number which ranges on the interval [0, 1]. This notion is equivalent to, and indeed has its origins in, that of fuzzy sets, originally introduced in [Zadeh 1965]; fuzzy sets intend to capture the notion of partial membership: the relation between an element and a set is given again by a real number between 0 and 1, where 0 stands for non-membership, 1 for complete membership and other values represent degrees of membership.

In the context of modelling agent reasoning capabilities, fuzzy logics present immediate advantages such as a easily defined map from human behaviour. It is also a computationally efficient solution for handling databases of knowledge of great size and/or when relation between concepts cannot be specified with certainty.

In [Geltkin & Arslan 2002] fuzzy logic is used to help agents reason about other agents' and precisely to produce a internal model theory for each of them, by observing their external behaviour.

Ambiguity was also subject of analysis in -for example- ambiguous communication models which tend to replicate natural language with this characteristic. Under the assumptions of truthfulness and trust, [Christof 1999] implement ambiguity in a Kripke model [Huth & Ryan 2000] for a multi-agent system; here the only action that takes place is communication of knowledge, which is effectively an update function.

Perhaps the most interesting application of fuzzy logic in artificial intelligence is that of a computational substitute for human emotions. Recent studies [Belavkin 2003] postulate the importance of emotions as a critical factor in the learning process efficiency, in turn motivated by neurological studies proving the equivalent for human beings [Damasio 1994]. [Belavkin 2003] simulates emotion as a noise factor and shows that high levels of noise enhance learning capabilities in early stages of the learning process.

With this as motivation, producing an "emotional" agent becomes a clear goal. An attempt to implement emotional behaviour by proposing a fuzzy logic model for it is given in [El-Nasr & Yen 1998]. Evidently this formalism would prove very useful if trying to handle very large or imprecise data, this goes beyond our intentions for this project. However it must be acknowledged that the ability to reason about other agents in the system, to the extent of creating a model for each other agent as in [Geltkin & Arslan 2002], has potential use in competitive games. The uncertainty of fuzzy logic however compels us towards refraining from adopting this model, since we hope to extract some final piece of information (namely the evolutionary fix point).

Possibilistic logics

"Possibilistic logic is a logic of uncertainty tailored for reasoning under incomplete evidence and partially inconsistent knowledge." [Dubois, Lang & Prade 1994]. Possibilistic logic extends LP syntax by allowing to append a real number belonging to [0, 1] to any statement; this number will stand for the certainty of the statement, i.e. how precise it is to deduce it from known data, or its possibility, i.e. how non-contradictory it is against known data. Intuitively this is closely related to fuzzy sets and fuzzy logic. One interesting aspect of possibilistic theory is that of preference settings for particular statements.

In the formalism adopted in this project (OCLP), preference is given explicitly by an ordering between subprograms; in possibilistic theory this is given implicitly by the degree of truth of a statement, where knowledge is represented by possibility

distributions over a certain variable, containing different degrees of truth which are compared to determine preference.

Dynamic logic programming

The fundamental feature of Dynamic Logic Programming [Alferes et al 1998] is that of program updating. The logical program updates, in every step (or change of state), following a second LP with generalised rules (allowing negation both in body and head). In programming agents, DLP can be very limited if a direct representation of belief revision is desired, since new information always overrides previous beliefs. Another limitation of DLP is its linear evolution of states, particularly when modelling speculative planning and reasoning about other entities' (agents') beliefs. This last restriction disappears in the generalisation of DLP: Multi-dimensional Dynamic Logic Programming (MDLP) [Leite et al 2000].

In [Dell'Acqua, Leite & Pereira 2001] a slight extension of MDLP is introduced and implemented to handle agents' belief revision (among other features).

2.4 OCLP

This formalism, introduced in [De Vos & Vermeir 2002], aims at the syntactic representation of decision making contemplating disjoint alternatives and an ordering between rules.

In OCLP the motivation is to combine both the idea of human choice among several options and the idea of preference over conflicting options.

As a general rule combining these would mean to follow the choice options as in CLP (see below) and whenever two rules are inconsistent, the one belonging to the most preferred (representing e.g.: higher importance, relevance or generality) component is followed.

2.4.1 CLP

The idea in CLP is to re-write datalog programs to an equivalent form so as to eliminate negation in the body of the rules. This is done by adding the symbol $\bigoplus$ to represent choice. That is, if in a datalog program we had

$$p \leftarrow \neg q$$
$$q \leftarrow \neg p$$

In a CLP, this will be re-written as

$$\mathrm{p} \bigoplus \mathrm{q} \leftarrow$$

According to [De Vos & Vermeir 1999] this new notation allows negation to disappear and the expressive power to remain the same as the previous one, "at least as

far as their (total) stable model semantics are concerned".
Formally a **choice logic program** is a set of rules of the form $H_r \leftarrow B_r$. Where H_r and B_r are sets of atoms, the ones in H_r assumed to be $h_{r_1} \bigoplus h_{r_2} \bigoplus ... \bigoplus h_{r_n}$, while the ones in B_r assumed to be $b_{r_1}, b_{r_2}, ..., b_{r_m}$, where the comma (',') stands for conjunction.

Some technical definitions

Extract from [De Vos & Vermeir 1999], the following definitions.
A **logic program** is a set of rules of the form $H_r \leftarrow B_r$. For each rule, H_r is called the **head** of the rule and B_r the **body**. Both are sets of atoms.
The **Herbrand Base** of the program, denoted β_P, is the set of all atoms appearing in the program.
An Interpretation I is a subset[1] of $\beta_P \bigcup \neg\beta_P$ such that if literal $a \in$ I, then $\neg\ a \notin$ I.
An interpretation is total if for each literal $a \in \beta_P$, either $a \in I$ or $\neg a \in$ I.
I $\bigcap \beta_P$ is called the positive part of I, denoted I^+. I $\bigcap \neg\beta_P$ is called the negative part of I, denoted I^-. $\beta_P \setminus I^+$ is called the positive complement of I, denoted $\tilde{I}$.

A rule r is said to be **applicable** if $B_r \subseteq I$. A rule r is said to be **applied** when, being applicable, it holds that the size of $H_r \bigcap I$ is 1. A rule is **satisfied** if it is applied or if it is not applicable. A **Model** for P is a **normal interpretation** that makes all rules satisfied. A model M is called **minimal** or **stable** if there is no other model N such that $N^+ \subset M^+$.

EXAMPLE 2. Jade's dinner. *Jade is about to consult with her partner, Edaj, what meal she would like. Their choices are pasta, rice, chicken and fish. Pasta and rice together are just too much. Jade does not like rice unless served with chicken. Edaj on the other hand, will "never ever eat chicken or fish", because she is a vegetarian. The situation can be represented by the following CLP:*

$$
\begin{aligned}
&pasta \bigoplus rice \leftarrow \\
&chicken \leftarrow rice \\
&\qquad\quad \leftarrow chicken \\
&\qquad\quad \leftarrow fish
\end{aligned}
$$

There is only one stable model for this simple example $\{pasta\}$.

2.4.2 OCLP

An OCLP is the collection of several CLP programs, called components, and an ordering between them, representing preference. Theoretically, the answer sets of a

[1] If A is a set, let "¬ A" denote the set {¬ a | a ∈ A}

component is inherited in the shape of interpretations from other less preferred components. Formally,

Definition 2.1. *An OCLP is a pair* $\langle C, \prec \rangle$ *, where C is a finite set of choice logic programs and "*$\prec$*" is what is called a strict partial order*[2]*. In this notation, if C1 and C2 are components, C1* $\prec$ *C2 means C1 is preferred over C2.*

Some inherited definitions

For an OCLP P, reformulation from [De Vos & Vermeir 1999]:
Define $P^* := \bigcup_{C_i \in C} P_i$, where P_i stands for the CLP corresponding to C_i.

Note how it is possible to have repeated rules, hence the need for a labelling function c which assigns to each rule of P^* the component where it lives. c(r) := component C_i s.t. r $\in$ C. Define the **Herbrand base** of P as $\beta_P := \bigcup_{C_i \in C} \beta_{P_i}$.

An **interpretation for P** is an interpretation for P^*.

EXAMPLE 3. Jade's sauce for her dinner. *Jade is about to cook dinner. She has no choice but to cook pasta, as seen before, but she is thinking about what sauce to have. She can prepare bolognese, carbonara or pesto, but no other things. She has already had pasta with pesto today so, if possible, she will opt for any of the other two. Now, it is well known that bolognese requires fresh tomatoes while cabonara requires cream. However Jade does not quite remember whether she has these ingredients or not. Her situation can be represented as the OCLP in Fig. 1.1, with P3* $\prec$ *P2* $\prec$ *P1.*

Example 3 clearly distinguishes between three levels of preference. Physical restrictions of the decision problem are stated in the most preferred component (P3), this is a standard in OCLPs as representation of a problem.
One level up, component P2 shows some preference by the entity which is deciding. Component P1 states default decision. Some interpretations: I = {tomatoes, bolognese}; J = {cream, carbonara}; K = {pesto}.

From the example above it can be observed how the notion of choice gains a new dimension with preference. It is possible to reason about the options available at each level of preference (component). Focusing on a component C and on an atom a, it is acceptable to talk about alternatives to a, be that in C, or in any component whose rules are preferred over the ones in C. This idea of alternatives turns out to be useful to describe some characteristics of OCLP.
Extract from [De Vos & Vermeir 1999]:

[2] A strict partial order is a transitive anti-symmetric anti-reflexive relation.

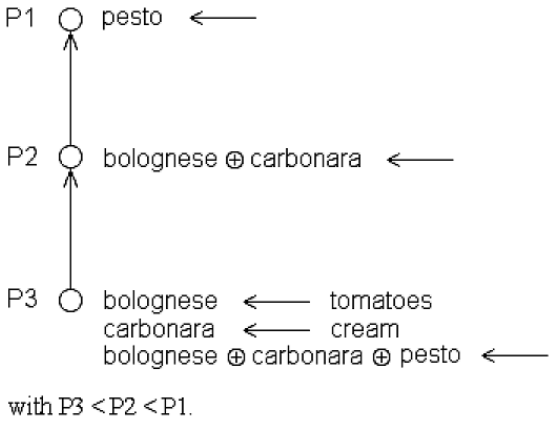


Figure 2.1: Jade's sauce for her dinner.

Definition 2.2. *If $P = \langle C, \prec \rangle$ is an OCLP and I an interpretation for it, with $C_1 \in C$. The set of* ***alternatives in*** *C_1* ***for an atom*** **a** *w.r.t I, denoted $\Omega^I_C(a)$, is:*

$$\Omega^I_C(a) := \{b \mid \exists r \in P^* c(r) \prec C_1 \wedge B_r \subseteq I \wedge a, b \in H_r \text{ with } a \neq b\}.$$

In example 3, alternatives for *pesto* in component P_1 w.r.t. interpretation K are {*bolognese, carbonara*}.

Credulous and Skeptical semantics

While giving means to decide upon conflict between rules belonging to different components, the formalism (so far) would seem not to determine what to do if two rules belonging to the same component are conflictive. Two immediate options arise: either discard the information they provide by disregarding both, or arbitrarily choose one of them, producing non-deterministic results.

The first option will be covered by what is called **skeptical semantics** while the second by **credulous semantics**, due to the nature of the method followed. Some more definitions are required, such as defeated rule and c-defeated rule.

In essence a defeated rule is strictly less preferred than another, conflictive rule, called defeater. A c-defeated rule, on the other hand, is a rule which is not more preferred (i.e. less preferred, equal, or not defined in the order) than the defeater. Formally, reformulation from [De Vos & Vermeir 2003]:

Definition 2.3. *Let I be an interpretation for an OCLP P. A rule $r \in P^*$ is* ***defeated*** *w.r.t. I iff*

$\forall a \in H_r, \exists r' \in P^*$ s.t. c(r') $\prec$ c(r) $\wedge B'_r \subseteq I \wedge H_{r'} \subseteq \Omega^I_{c(r)}$(a),

where the r' rules are called ***defeaters****.*

Definition 2.4. *Let I be an interpretation for an OCLP P. A rule $r \in P^*$ is* ***c-defeated*** *w.r.t. I iff*

$\forall a \in H_r, \exists r' \in P^*$ s.t. $\neg(c(r) \prec c(r')) \wedge r' is applied w.r.t. I \wedge H_{r'} \subseteq \Omega^I_{c(r)}$(a),

where the r' rules are called ***c-defeaters****.*

Definition 2.5. *In OCLP, an interpretation is a* ***skeptical model*** *if and only if it makes all rules in P^* either: non applicable, applied, or defeated. It is a* ***credulous model*** *if and only if makes all rules in P^* either: non applicable, applied or c-defeated.*

OCLP in Multi-agent Systems

As suggested by [De Vos 2001] OCLP can be intuitively used to represent both knowledge and reasoning skills in an agent. [De Vos 2001] proposes that agents will receive information, combine it with their OCLP program, compute their answer sets and retransmit them to other agents via uni-directional communication channels. Notice that conflictive input should be discarded. [De Vos 2001] identifies two arbitrary agents: the input agent (the one who starts the process) and the output agent (the one whose answer set is the output of the system) and is therefore not allowed to have outgoing communication channels. [De Vos 2001] also identifies two well-distinct cases: the simple case -without cycles- means that every communication channel is used once and when all communication has taken place, the answer sets for the output agent is taken. A second, more complicated case class arises if cycles are present; here the system undergoes an evolution of states and should eventually stabilize to give what is then considered the solution (a consensus between all agents after a long debate). Formally:

Definition 2.6. *An* ***Ordered Choice Logic Programming Agent System (OCLPAS)*** *is a tuple $\langle A, I, O, C \rangle$, where*

- A is a set of agents specified by OCLPs
- I $\in$ A is the input agent.
- O $\in$ A is the output agent.
- C stands for the communication channels represented by an anti-reflexive relation, such that:

$\forall$ a $\in$ (A $\setminus$ I), $\exists$b $\in$ A, a $\in$ C(b) and
$\forall$ a $\in$ (A $\setminus$ O), $\exists$c $\in$ A, c $\neq \emptyset$ with c $\in$ C(a)

Agents of this shape will, however, produce different outputs at different times. This applies also to the output agent, meaning the system will have several outputs. It is possible to hold all of the information regarding each step in the evolution of the system (by "step" here we mean a message-sending act) by recollecting all messages passed. In logic terms this means classifying under some label the subset of literals belonging to the output of all agents. In order to do this, some concepts must be defined.

More inherited definitions

Definition 2.7. *Suppose F is an OCLPAS* $\langle$ *A, I, O, C*$\rangle$ *, the* ***Herbrand Base*** *for F, denoted* β_F*, is the union* $\bigcup_{a \in A}(\beta_a)$.

Definition 2.8. *Suppose F is an OCLPAS* $\langle$ *A, I, O, C*$\rangle$ *, an* ***interaction profile*** *for F is a function:*

S: A $\rightarrow \beta_F \bigcup \neg\beta_F$

Definition 2.9. *Suppose that* a $\in$ *A and that S is an interaction profile for an OCLPAS F, define* ***output of*** **a** *with respect to S, denoted* $Out_S(a)$*, as*

$Out_S(a) :=$ S(a).

Definition 2.10. *Suppose that* a $\in$ *A and that S is an interaction profile for an OCLPAS F, define the* ***input of*** **a** *with respect to S, denoted* In^a_S*, as the union of outputs of the agents which send information to* a*, i.e.:*

$In^a_S := \bigcup_{b \in A, a \in C(b)}(S(b))$.

S(a) will represent the information output of agent 'a'.

Given this, one would like to describe what an agent is supposed to do with incoming information, particularly with inconsistent information. In this matter [De Vos 2001] defines a **filtered input**, as the literals in the incoming information which are not contradicted by any other. For example if agent Jade receives the messages $\{theskyisblue, applesarenice, \neg theskyisblue\}$. The filtered input for Jade will be $\{applesarenice\}$.

Definition 2.11. *Suppose that* a $\in$ *A and that S is an interaction profile for an*

OCLPAS F, the ***filtered input*** *of an agent A, denoted* $In_S^f(A)$ *is*

$$In_S^f(\mathrm{A}) = In_S^{A,+} \setminus (In_S^{A,+} \bigcup In_S^{A,-}).$$

This filtering procedure is safe but restrictive, since we have no means of distinguishing between different sources hence becoming impossible to define a preference among them (a feature for which OCLP is ideal).

[De Vos 2001] goes on and proposes a way of handling information: updating the internal program. Adding a new component C which is less preferred than any other component already present. This new component will have statements asserting incoming positive information. Formally,

Definition 2.12. *Suppose* C_A *is the OCLP corresponding to an agent A and that U is a set of atoms with* $U \subseteq \beta_F \bigcup \neg\beta_F$ *. An* ***updated version*** *of A with respect to a set of atoms U, denoted* A^U*, is a program that extends* C_A *by adding an extra component P with the following properties:*

1. $\forall$ a $\in$ U, "a $\leftarrow$" $\in$ P
2. $\forall$ C $\in C_A$, P $\prec_{A^U}$ C
3. $\forall C_1, C_2 \in C_A, C_1 \prec_A C_2 \Rightarrow C_1 \prec_{A^U} C_2$

The final idea which needs to be mentioned here is that of how systems evolve. It is of our particular interest to identify when the system has reached an evolutionary fix point. That is when the output of every agent no longer varies.

2.5 JADE

2.5.1 Introduction

In [Bellifemine, Poggi & Rimassa 2001] JADE is officially introduced and detailed description of its features are presented. JADE (Java Agent DEvelopment framework) is a software package containing programs to run, and a collection of classes and other tools to aid in, the development of multi-agent systems in Java. It abides by the FIPA [Vidal 2003] standards.

The idea behind JADE is to automate all standard tasks that take place in a multi-agent system leaving the programmer to code the essence of every agent (the agent behaviour) and to specify communication features.

JADE is also a distributed platform as it can handle agents running in several instances of the application in a machine or in several interconnected machines; all of these remaining transparent to the programmer.

2.5.2 Inside JADE

Containers and platforms

In JADE the world is divided into Platforms: essentially a collection of running processes, called Containers. A container is an active process in JADE (in practice it is a Java Virtual Machine -or JVM-), agents are essentially passive -in the sense they are only adjudicated run-time by JADE- and live in one container which handles all of its agents' life-cycles, by adjudicating runtime to whichever behaviour needs to be executed.

For each platform there is a **main container** (or **front-end**) and an arbitrary number of other containers. Each one of them registers with the main container when initialised. The main container therefore holds a list of registered containers in the current platform as well as a list of the agents' addresses (a GADT - **Global agent descriptor table**).

Local versions of this table (LADT - **Local agent descriptor table**) are present in every container. The main container performs the inter-platform re-routing task for communication by acting as a IIOP server listening to the official platform ACC (**Agent communication channel**) address. In the main container, two mandatory tool-agents are run, including: the default DF (**Directory Facilitator**) agent and the AMS (**Agent Management System**).

Directory facilitators are there to record a logical subset of agents and advertise them by services they provide. The AMS on the other hand is unique per platform and its task is to provide platform methods to handle agents, such as `suspendAgent()`.

Message delivery subsystem

Since this is all handled transparently to the programmer and for the purposes of this project, the internal functionality will only be mentioned. Basically, JADE codifies agents GUIDs (or **Global Unique IDentifiers**) according to FIPA specifications; with respect to communication means. JADE distinguishes three of them:

1. Java events - (an ACLMessage object is passed) if sender and receiver live in the same container.
2. RMI - (serialization/deserialization of the message and an RMI call) if sender and receiver live in different agents on the same platform.
3. CORBA IIOP - (message is coded/decoded to/from IIOP protocol) if sender and receiver live in different platforms.

Some tools provided

Due to the high level of abstraction of the JADE platform, a collection of high-level monitoring/debugging tools is provided, in the shape of FIPA agents; these have no

special privileges except they rely only on AMS to perform their tasks.

RMA or **Remote Management Agent** is one of these, providing a GUI which allows the user to directly interact with the AMS and moreover to monitor changes in the state of the platform. With this, the user is granted access to the AMS methods and hence can create agents, destroy agents, suspend agents, etc.

DF or **Directory Facilitator**, aforementioned, has also a GUI interface and its methods can be accessed by the user as well, to add/remove agents and configuring their advertised services.

DummyAgent is a tool providing the user with means to send arbitrary messages, which turn out to be extremely useful in testing/debugging.

Sniffer Agent, on the other hand, is a passive agent which has access to the communication channels of any other agent and therefore allows the user to monitor messages sent to and from a list of "sniffed" agents in the platform.

A JADE agent

Since FIPA specification states no restrictions whatsoever on internal agents' structure, JADE leaves little restrictions as well, these being: computational time and Java expressive power. In [Bellifemine, Poggi & Rimassa 2001] multi-agent systems' needs (discussed in Section 1) are presented along with the design solutions JADE implements, as follows:

Theoretical requirement	**Design solution**
Agents are autonomous	**Agents are active objects**
Agents are social	**Intra-agent concurrence is needed**
Messages are speech acts	**Asynchronous messaging has to be used**
Agents can say 'no'	**Peer-to-peer communication model is needed**

JADE balances these necessities with computational power restrictions by allowing each agent a single Java thread. In JADE, the different tasks an agent can perform are abstractly represented by **Behaviours**. Each behaviour represents one of the agent functions.

Thus in writing an agent, the programmer must extend the Agent class provided by JADE and create the desired number of subclasses for each desired Behaviour. So that JADE knows how to operate and handle every given behaviour, a predefined class hierarchy is facilitated to the programmer, where the top class is Behaviour; inherited from this: SimpleBehaviour and ComplexBehaviour. In turn, from SimpleBehaviour we have OneShotBehaviour and CyclicBehaviour; while from ComplexBehaviour we have SuquentialBehaviour and NonDeterministicBehaviour.

SimpleBehaviour is the kind of behaviour that will be called occasionally and will obtain the absolute control of the agent thread until it finishes. Its subclass OneShotBehaviour will execute only once, never again and CyclicBehaviour which executes repeatedly, never finishing.

ComplexBehaviour accounts for time expensive processes which ought to be split

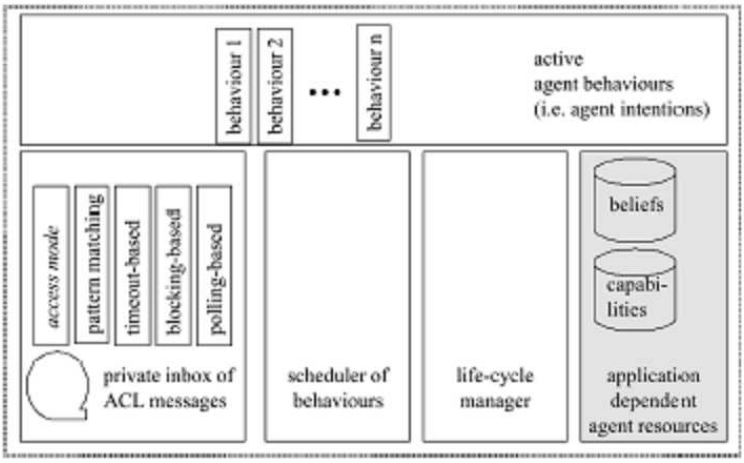


Figure 2.2: JADE agent architecture.

in several sub processes, hence ComplexBehaviour holds a list of children Behaviours and executes them, but allowing to be interrupted in between any of those. Its subclass SequentialBehaviour performs this execution in a defined sequence and stops when the last one is finished; NonDeterministicBehaviour, on the other hand, allows itself to be terminated at any stage arbitrarily and follows an apparently chaotic (a round-robin algorithm is used) sequence for its children.

The concise facts to highlight from this for our purposes in this project are:

- Agents run simultaneously, support given by Java multi-threaded capabilities
- Messages are handled asynchronously

In developing a JADE agent, the focus is therefore in identifying the different behaviours, their types and coding them in Java language. Other agents features discussed in Section 1, such as agent beliefs are left to be programmed independently. Figure 2 displays clearly what JADE handles and what is left for the programmer.

2.5.3 Alternatives to JADE

Agent programming and multi-agent systems, as stated in Section 1, have emerged relatively recently and gained attention only in the last ten years; thus it is not surprising that the software implementation field for MAS is still immature and standards for unification are yet to be established. [Bellifemine, Poggi & Rimassa 2001] make reference to some of the recent efforts, such as DMARS [Rao & Georgeff 1995], RETSINA [Sycara et al 1996] and MOLE [Baumann et al 2003] as well as to their standardizations: KSE [Patil et al 1992], OMG [Milojicic et al 1998], and FIPA [Vidal 2003].

FIPA being the predominant standard and JADE its most popular implementation, JADE is chosen as the tool for this project.

2.5.4 Why not JADE

While supporting extensive features over the needs for this project, two key imperfections are to be mentioned. The first is JADE messaging system, which uses a single message queue per agent, this is rather conflictive with multiple behaviours for simultaneous asynchronous communication, but only result in a computational limiting factor, hence not a major problem for the purposes of this project.

The second is to do with robustness: even though JADE is a distributed application itself, its reliability is rather dependant on key components, specifically the front-end container and the AMS agent. However, due to the static nature of the communication channels in this project, JADE's message caching should allow the system to carry on even in the event of a failure in the GADT.

2.6 OCT

OCT [Brain & De Vos 2003] is an front-end for answer set solvers developed by Martin Brain. It will take as input an OCLP program with the following syntax.

An OCT input program structure.

```
component C01 {
rule11
rule12
...
rule1n
}

   component C02 {
rule21
rule22
...
rulem
}
...
component Cxx {
rulexx1
rulexx2
...
```

rulexx3
}

[Preference relations]

Where each rule is of the form "a1 + a2 +...+aq :- b1, b2,...,br" and [Preference relations] are lines of the form "C0x < C0y".

It will return the answer sets. Internally it will produce a LP that will be then solved by Smodels. For the purposes of this project OCT will be used to solve the OCL programs of each agent.

2.7 Conclusion

Using OCLP to model the reasoning capabilities of an autonomous agent grants an easy way of representing the human ability to choose among disjoint alternatives as well as defining preference among potentially conflictive situations. In parallel, JADE provides the entire infrastructure for standard multi-agent development and OCT gives an automated solution for running OCL programs.

In the remaining part of this dissertation we approach the technological alternatives and proceed to develop the tractable OCLP agent definitions (TOADs), a collection of Java classes as extensions to the JADE agent definitions. We develop an interface for agents to communicate with each other by means of an ontology and a protocol for information interchange. Furthermore the agent cycle is made precise and OCT is integrated so as to compute the answer sets. Finally we analyse the behaviour of the OCLPAS (Ordered Choice Logic Programming Agent System) and observe the correlations with Game theory.

Chapter 3

Requirement analysis and specification

3.1 Introduction

As suggested in [Somerville 2001], for the requirements analysis it is our purpose to identify potential user needs in abstract terms first and then, based on these, to analyse the concrete system requirements. Therefore, in the first part of this chapter we define the features and limitations of the product from the user's point of view. Consequently we analyse in more detail the requirements of the system and propose in each case one or more feasible solutions. As a conclusion we dictate the specification: our selection of solutions together with a justification for our choice.

3.1.1 Scope of the product

The product is meant to cover two kinds of users: the programmer and the researcher. We intend to give the programmer who wants to incorporate OCLP to the reasoning capabilities of his/her agent, a tool to render automatic the lifecycle process and intercommunication with other agents of the like. For this, the scope of the product is restricted to run the lifecycle providing the programmer with interruptions for the execution of his/her own code as well as events which fire at relevant stages. It is also our purpose to create a framework for experimentation with purely OCLP-minded agents, giving an accessible way to define, add, remove and edit agents whose only functionality is that of updating and communicating. For the latter we intend to provide a simple interface to ease the operability of the system. Here the scope is slightly more extensive as it will also cover the design and implementation of an auxiliary agent (called Queen agent), through which the user may interface with the system.

3.1.2 Definitions, acronyms and abbreviations

Throughout this chapter we shall make reference to ontology classes by their respective names (introduced in Section 3.3.2, page 51). We will be referring to the collection of classes and auxiliary agents that we will develop as either: **the program**, **the system**, **TOADs**, **the application** or **the framework**. When speaking of **the user**, we mean either the person (if running in retained mode) or his/her program (if running in runtime mode). Specific terminology regarding logic programming in multi-agent systems will include:

- **OCLP**: Ordered choice logic programming
  - **Internal OCLP program**: the program included in each agent, which defines her knowledge and reasoning abilities.
  - **OCLP updating process**: the process of appending components to the internal OCLP program as described in Chapter 2, Section 4.2, page 23.
  - **Solving an OCLP**: finding the stable models of a given OCLP program.
- **OCLPAS**: Ordered Choice Logic Programming Agent System.
- **Input agent**: the agent which starts the communication domino effect.
- **Output agent**: the agent whose output is interpreted to be the output of the system.

Terminology defined by TOADs:

- **Retained mode**: an agent is said to be running in retained mode when she is running as described in the experimental framework solution, in Section 3.1.1.
- **Valid inputs** and **valid outputs**: each agent has an internal list of agent names from which she will accept messages (the valid input) and a list of agent names to which she will broadcast her message.
- **Communication channels**: the collection of valid inputs and valid outputs for all agents in the system.
- **Current step or cycle**: for each agent, an incremental counter which represents the number of times the agent carried out the listen-update-calculate-broadcast process, each of those called a step, or a cycle.
- **A project**: the information regarding all the agents in the system, including their original internal OCLP program, the semantics specification and the communication channels information.

3.2 User requirements

The system should be able to compute the OCLP-lifecycle of each agent. It should handle all protocols, serialization and de-serialization of concept objects transparently to the user.

3.2.1 User requirements retained mode

1. The program must allow the user to add, remove or modify any of the agents in the multi-agent system.
2. The program must be able to take text input to define the OCLP programs of each agent.
3. The program must allow the user to define the communication channels between agents.
4. The program must allow the user to determine which semantics should be used by each agent.
5. The program must allow the user to run a single step of the system or to run indefinitely until a fix-point is found and retrieve it.
6. There must be an option to load/save a current project.
7. There must be options to import/export parts of the system from/into relevant file formats.

3.2.2 User requirements runtime mode

1. The program must give means for the programmer to alter all of the relevant information of the local agent both in a high level and a low level fashion.
2. The program must provide events that fire upon relevant stages of the OCLP agent lifecycle.
3. The program must provide a special event representing the achievement of the evolutionary fix-point.

3.2.3 Assumptions and dependencies

Assumption 1: the user will be expected to be familiar with the OCLP formalism and with the OCT syntax. Assumption 2: the user will be responsible for spelling mistakes while defining atom names, case differences will be interpreted as different atoms. Assumption 3: the user will be expected to handle the interruptions provided with responsibility. Interruptions will effectively stop all processes in a TOADs agent,

and failure to return control to the superclass may imply the effective death of the agent. Dependence 1: the system will attempt to run OCT. It is the user responsibility to install and compile OCT (or any alternative answer set solver with identical syntax) and make it runnable from any directory.

3.3 System requirements

3.3.1 Functional requirements

- Req. 1, INPUT:
  - Req. 1.1: An OCLP program as a string satisfying OCT syntax.
  - Req. 1.2: A Boolean value to determine whether credulous or sceptic semantics will be used.
  - Req. 1.3: Two lists of agent identifiers representing the valid input and the valid output (i.e. the communication channels).
  - Req. 1.4: An instruction at the creation of the agent, specifying whether to run in retained or in runtime mode.
  - Req. 1.5: At each step, a TOAD agent will require a message containing one or more answer sets from each of its valid inputs.
- Req. 2, OUTPUTS:
  - Req. 2.1: At each cycle, a TOADs agent must be able to return those answer sets which are solutions to the program obtained by the OCLP updating process with respect to the input received.
  - Req. 2.2: The program must be able to evolve automatically and identify the evolutionary fix-point of the OCLPAS.
  - Req. 2.3: The program must be able to log messages passed in a human readable format.
- Req. 3, STORAGE:
  - When running in retained mode, the system must be able to save a project into a file and restore a project from a file.

Input requirements 1.1 to 1.3 represent the essence of a TOADs agent: her knowledge, reasoning skills and her social skills. In retained mode these have to be easily accessible to be edited by the user. Either a text or a simple graphic interface should be provided. See section 3.3.1 about interfaces. Input requirement 1.5 means that we ought to:

- Define an ontology for a common understanding and standarised description of the structure of the information passed. See Section 3.3.2 for ontology.
- Define a content language. Luckily this is provided by JADE. See Section 3.3.2 about Protégé.

Output requirement 2.1 means we need: a way of updating the internal OCLP program depending on incoming information, which we shall design and develop, and an answer set solver (see Section 3.3.6) for which we use OCT.

Output requirement 2.2 means we have to ensure the system can run smoothly without any need for user input once the initial input has been given. We shall achieve this by designing a lifecycle and a protocol which allow for all agents to synchronously interact. Furthermore we shall design TOADs agents to be able to survive and handle exceptions, external errors and other characteristic problems of distributed systems, such as latency or disconnection. See Section 3.3.2 for more details.

Storage requirement means that for each agent, OCLP program, semantics choice and communication channels should be saved and restored from files. For this, we need to define a file format. Alternatives are:

- Object serialization: the objects representing this information for each agent are serialized and saved to disk. This has the advantage of simplicity and compactness.
- Textual representation: information is saved as strings, where a specific syntax is designed and followed. This has the advantage of readability and alterability.

The Queen agent has access to all the information needed and can therefore perform this task. The Queen agent has also got the power to alter every piece of information in the system; hence it can perform the restoring operation as well. Alternatively this feature may be programmed locally at each agent. The user must be allowed to specify if the system should be saved as a whole (in a single file) or every agent (in separate files) individually. There must be an option to import and export OCT compatible .oclp files.

3.3.2 Non-functional requirements

Usability - interfaces

In TOADs, the user interface is included in the auxiliary agent named **Queen agent**. This agent will be capable of ordering agents to perform a cycle, update their information and retrieve their information; hence giving the user a full interaction capability with all agents running in retained mode which were locally told to obey to the Queen agent in question.

If running TOADs in runtime mode, i.e. extending the classes provided, the interface would require:

- A logger tool which can write information in a human-readable format regarding message passing, knowledge base and evolutionary status for every agent. This logger must be embedded in the TOADs definitions so as to maximise efficiency and to guarantee autonomy.

  If running TOADs in retained mode, be that for dummy agents or extended user-defined agents, the interface needs to be more complicated, requiring:

- A way of displaying the internal OCLP program of any agent on the screen in a human readable format.

- A way of allowing the user to alter this text and uploading the updated version back into the agent.

- A panel displaying the communication channels and means to change them

Note that while running in retained mode the logging tool will still be present. Furthermore it is required that the user is allowed to specify whether credulous or sceptic semantics should be used. In runtime mode this feature will come as a simple method while in retained mode it must be part of the visual interface.

Robustness

If executed in retained mode, the system is expected to be run on a single computer, where errors are local and therefore traceable by the user. However it might be the case that an experiment is carried out between several platforms, maybe not all of them supervised by an administrator. It might also be the case that TOAD is executed in runtime and expected to be robust. In distributed systems, robustness extends beyond the integrity of the application alone. For an agent to be robust, she must be programmed to expect other agents to disappear or stop responding without a traceable reason. For this reason, TOADs should be able to handle this sort of exceptions. Possible solutions would involve one or more of the following:

- A time limit for message listening.

- Error events which fire when these occur.

In TOADs there are other things that can go wrong. Message filtering can be designed to only accept ontology and protocol matching messages. However, improperly written or malicious agents could embed erroneous objects in a properly composed message. TOADs must be able to catch exceptions at this level and inform the user.

Interoperability means that external factors such as the operative system or OCT itself could produce errors which are merely circumstantial, i.e. since the system could potentially be running in one of many operative systems, domain specific exceptions must be anticipated. TOADs must be able to survive this kind of events by catching the exceptions and handling them. Alternatives include:

- Leaving it up to the user to handle a certain error.
- Retry to execute the relevant piece of code several times.
- Encoding the error as part of the AnswerSets output.

Portability

TOADs agents and the auxiliary agents should be able to be run in any computer capable of running JADE and OCT. For requirements which arise as a consequence of these constraints, see subsection about implementation.

Standards

The program is required to adhere to the FIPA standards.

Implementation

So far the use of JADE was assumed. The challenge of demonstrating the use of OCLP in multi-agent systems means we need a multi-agent framework: a message handler and a process coordinator for the diverse agents and their behaviours are crucial. Choosing JADE meant that all the message passing was handled transparently, agents can be implemented in a high level programming style by defining their behaviours and reactions, and that all of the FIPA standards were automatically adhered to. Drawbacks from this choice include the facts that the agent architecture must be put in terms of behaviours and that the imperfections of JADE (see Section 2.5.4 Why not Jade) would be suffered.

The choice of JADE immediately forces the development of the application in the Java programming language.

Interoperability

Each TOADs agent must be able to interact with other TOADs agents and with the Queen, clearly. As described in Section 3.6, TOADs will use a third-party front-end called OCT to solve OCLP programs. In order to interact with OCT, TOADs is required to

- Produce OCT's valid input
- Parse OCT's output results
- Parse OCT's error messages

TOADs must be able to run OCT and wait for its output. This can be programmed to happen both synchronously (i.e. the agent will run OCT and wait for an answer

before continuing) or asynchronously (OCT output is handled in a parallel thread and firing an event when finished).

TOADs should make it clear to the user what command line will be executed to run OCT, so that the user may install and compile OCT where appropriate.

Ontology

Following JADE standards and in order to give a basis for the representation of the different objects and functions in the world of TOADs, an ontology has to be explicitly defined. An ontology can be given independently from JADE as a class hierarchy of our own, it can be written following JADEs ontology class standards by hand, or it can be generated by using Protégé [Prot[́e]g[́e]].

For the ontology definitions we can identify three main subsections:

1. OCLP programs
2. Message passing
3. Knowledge base

The ontology for OCLP programs needs to be structured so as to declare a hierarchy. The main class being the OCLP program, it should contain an arbitrary number of components and an arbitrary number of relation rules.

The components should be split into rules and these into set of atoms of the same class that that of knowledge base. Regarding message passing, the ontology should define encapsulating classes for the following messages:

- Answer sets broadcast. It is necessary for each agent to produce a message whose contents structure is well defined. It should contain information about:
  - Answer Sets. The crucial information carried.
  - Evolutionary Status. Needed to determine whether an evolutionary fixpoint has been achieved.
  - Cycle identification. Required to assure synchrony.
- Queens information request and retrieval messages. These must come wrapped in an ontology class which must contain the type of information requested. The Queen may request the agent to retrieve:
  - Its OCLP program.
  - Its communication channels description.
  - Its default semantics (Credulous/Skeptic).
  - Its last sent Answer Sets object.

– Queens information retrieval. The message containing the information requested, which should encapsulate:
– The object requested.
– Current cycle id.

3.4 Requirements specification

– The application shall be developed in JAVA 1.3 or higher since JADE framework requires so.
– The application shall be developed under JADE so as to adhere to the FIPA specification.
– OCT will be used as the answer set solver. OCT is an active project and several improvements are planned for the near future. We can take advantage of these by interfacing with OCT.
– A special auxiliary agent called "The Queen" will be developed to act as an interface between the human user and the MAS. The Queen agent will come with a very simple graphical interface, this interface is a prototype and was not intended to be final or to abide by any HCI standard.
– Events shall be programmed to fire upon exceptions, errors and connection time out.
– An ontology shall be defined for the representation of OCLP programs, answer sets and message structures.
– Protégé will be used to develop this ontology and a suitable content language.
– There were no documentation requirements.

Chapter 4

Design

4.1 Introduction

Following FIPA [Vidal 2003] specifications and working under the JADE framework, it is obvious to seek a behavioural decomposition architecture. Decisions have to be made regarding how to split the system into behaviours.

In designing TOADs, an explorative approach is used, meaning that, after a high level decomposition of the system into sub-systems, we begin by defining our data structures (our ontology) and proceed from there to the design of the interfaces between the sub-systems. When subdividing our system we use the GAIA [Wooldridge 2002] methodology, we identify roles and for each role, its characteristics: responsibilities, permissions, activities, protocols.

We pay particular attention to the design of the Queen agent and its GUI.

4.2 TOADs as a Client-Server framework

Since TOAD agents running in both retained and runtime mode can interact with agents running in both modes alike, the TOAD framework can be seen as a Client-Server framework, where Queens act as servers and ordinary agents as clients, in a fat-client fashion (see Figure 4.1). The protocols for all the kinds of interactions will be described later.

4.3 Roles

In the remaining part of this chapter we introduce the concept of roles, identify the various roles in TOADs and design an ontology. A legend for the diagrams in this chapter can be seen in Figure 4.2.

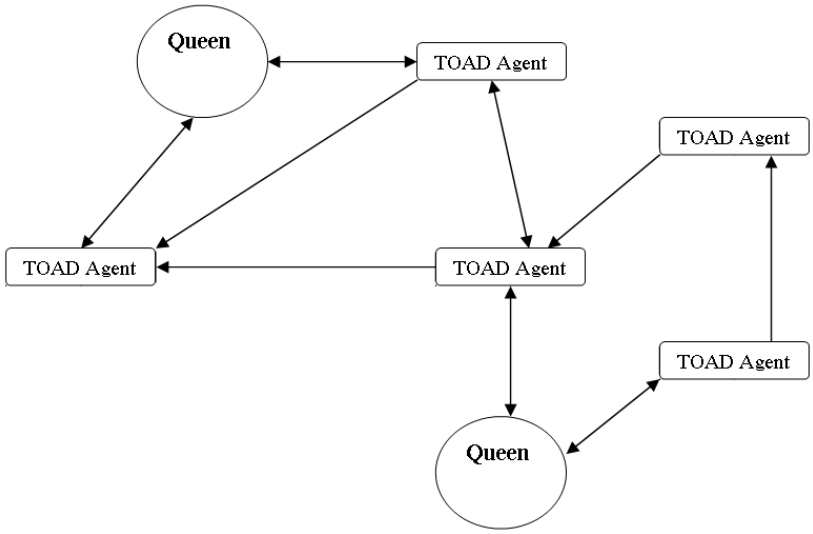


Figure 4.1: TOADs as a Client-Server framework

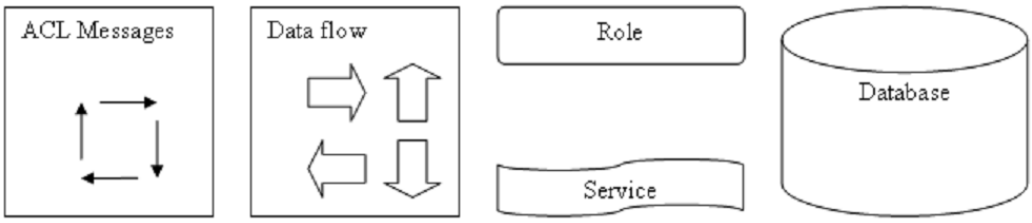


Figure 4.2: Legend for design diagrams

4.3.1 What is a role?

In the GAIA methodology, roles are an agent system equivalent to society roles. It is a collection of behaviours that an entity is expected to adopt by other entities interacting with it, then that entity is said to assume a certain role. For example, in a building, porter and cleaner are two roles; people expect the porter to watch at night and they expect the cleaner to keep the building clean. It may be the case that a single entity (a person or an agent) assumes more than one role; for example the same person doing the night-watch could also clean the building during the day.

Roles are defined by four attributes: **responsibilities**, **permissions**, **activities** and **protocols**.

Responsibilities are the key aspect in a role, as they describe it for external observers; they can be split into two categories: **liveness properties** and **safety properties**. Properties of the first type include the results a certain role is supposed to bring about, the positive goals, what has to happen. Properties of the second type include states that must be avoided, what must not happen. For example, a porter's liveness properties include to welcome any person living in the building, whereas a cleaner's safety properties include to avoid letting the building condition go below hygienic standards.

Permissions represent the security clearance a certain role has, what parts of the system it is allowed to interface with and how. For example, a porter may have access to the alarm activation both for activation and deactivation, whereas the cleaner wouldn't; on the other hand the cleaner may have access to the cleaner's locker where cleaning tools are kept.

Activities are those tasks a certain role performs inside its domain and which remain transparent to other roles. This may interfere with the system but need no interaction with other roles. For example, cleaning a window is an activity for the cleaner, whereas locking the building is not for the porter as s/he needs to communicate with, for example, the cleaner to avoid locking him in.

Protocols define how roles interact with each other. Following the previous example, a do_you_live_here protocol could be used by the porter to interact with other people who want to come into the building.

4.3.2 Roles in TOADs

In TOADs there will be two kinds of agents: the Queen agent and TOAD agents: the first kind is the one to be operated by the human user, the latter is the one which provides the required functionality. This distinction was made in order to make TOAD agents reusable without the need for a human supervisor, hence TOAD agents can be implemented with the aid of a Queen agent and then run independently (in what we call **runtime mode**).

Within each of these two agents, we identify several roles. The motivation for this subdivision is a transparent distinction and the reusability of the various services

included in each agent. I.e. if new agents were to be developed where the main TOAD engine is not required, but interaction with TOAD agents is needed, the given roles give a clear way of interacting with an agent knowing that the role's permissions ensure no other sections of the agent are altered. Furthermore, as we shall see later, roles can be reused in the case a programmer wishes to develop an agent using only some of TOADs' services.

For example, a TOAD agent can become **subscribed** to another agent so as to periodically send it information regarding its state. A programmer may want to develop a similar agent and decide to make it interact with the Queen agent in this way; in this case a transparent description of a **subscriber** role is certainly needed.

Subdividing the Queen agent into several roles also means that some of them can be reused, for example, in other server-like applications which may need to command TOAD agents, or just supervise a single agent.

Agent supervision and the Subscription protocol

Whenever the user requires that a particular agent is supervised, a way is needed to command the agent to periodically send updates of its internal state to the Queen, or to stop doing so. Therefore there should be a special role in charge of handling these kind of commands and there should be also another role (to be assumed by the Queen) to process incoming information that was not explicitly requested, under this protocol. While originally the intention was to design a Subscriber role which would handle all stages of a subscription (setting up, cancelling, sending information), in the final version of TOADs it will only handle setup and cancellation; this is due to the fact that it is effectively pointless to redirect the control flow to a different role in the middle of the execution of the main engine role (where it should be summoned to send the update message). Instead the role will simply update the internal state of the agent and the main engine role will take care of the rest.

4.3.3 The TOAD agent – Roles

Within TOADs we identify the following roles for TOAD agents:

OCLP Agent (main engine)

- Responsibilities:

  1. to ensure synchrony in sending and receiving OCLP_message messages to and from other TOAD agents.
  2. to ensure robustness by setting a deadline for messages to come.
  3. to reply with REFUSE performative to those messages which arrive out of sequence.

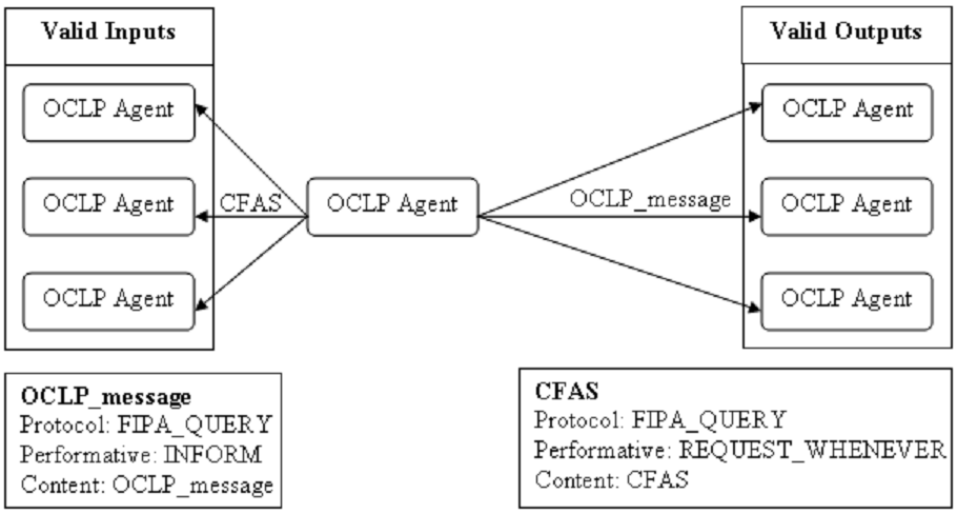


Figure 4.3: OCLP Agent (main engine) role.

4. to run an OCLP agent cycle preparing the answer sets for the updated program.

- Permissions: full access to the agent's public interface
- Activities:
  - to broadcast the Call For Answer Sets (CFAS).
  - to filter the input, ignoring conflicting information.
  - to update the internal OCLP program using the filtered input.
  - to interact with OCT to obtain the answer sets.
  - to store/send the answer sets to the valid output.
  - to identify the evolutionary fix-point.
- Protocols: this role shall communicate under the protocol FIPA_QUERY, as shown in Figure 4.3.

Information Facilitator

- Responsibilities: to answer information queries from the Queen.
- Permissions: read access to the internal information in the agent.
- Activities: To answer information queries from the Queen.
- Protocols: this role shall interact with:

  - the Information Retriever − using FIPA_QUERY, as shown in Figures 4.4 and 4.5.
  - the Subscriber − using FIPA_SUBSCRIBE, as shown in Figure 4.6.

Information Updater

- Responsibilities: To update the agent info upon receiving the update message from the Queen.
- Permissions: full access to the agent's public interface.
- Activities: To update the agent info upon receiving the update message from the Queen.
- Protocols: FIPA_REQUEST, as shown in Figures 4.4 and 4.5.

Retained Mode Handler

- Responsibilities: To handle incoming request from the Queen, e.g.: to make a single step, to switch to runtime mode
- Permissions: read/write access to the fields containing the last output prepared by the OCLPAgent role and the field which determines whether the agent is running in retained mode or in runtime mode
- Activities:
  - if a STEP command arrives and the stored output is up to date, send the stored output; if it is out of date, reply to the Queen with a FAILURE message.
  - if a RUN command arrives, switch to runtime mode.
- Protocols: FIPA_PROPOSE, as shown in Figure 4.4.

Subscription Handler

- Responsibilities: to maintain the subscription status of the agent.
- Permissions: access to the boolean field representing whether the agent is subscribed or not.
- Activities: change the subscription field to true if a Q_setupSubscribe message is received or to false if a Q_cancelSubscription message is received.
- Protocols: FIPA_SUBSCRIBE, as shown in Figure 4.6.

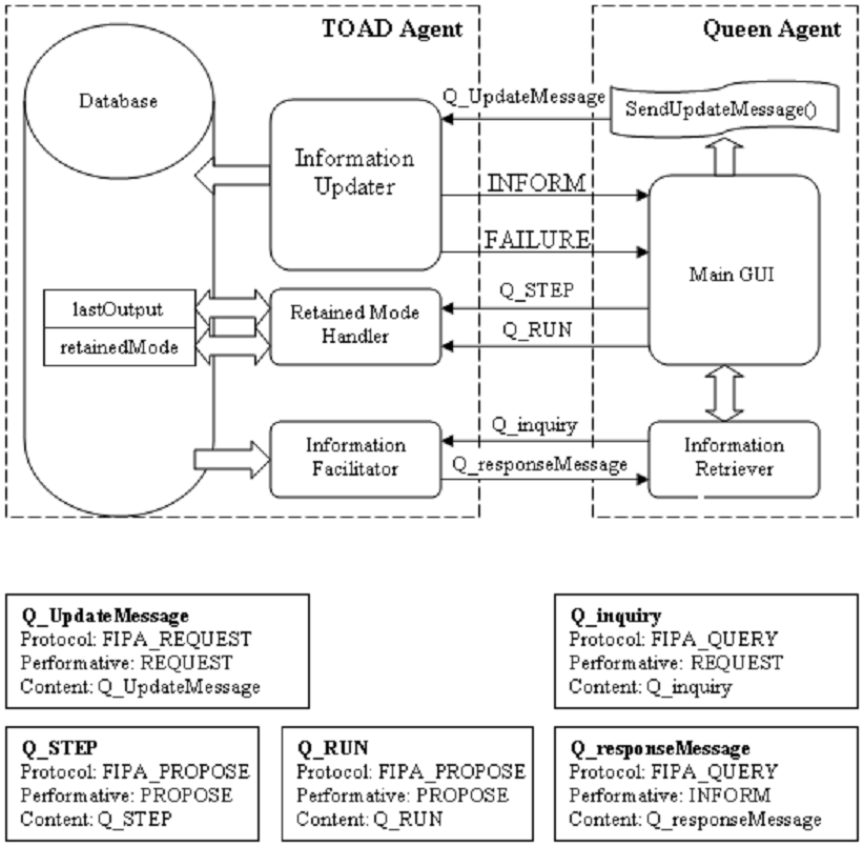


Figure 4.4: Main GUI and related roles.

4.3.4 The Queen agent – Roles

We also identify the following roles for the Queen agent:

Main GUI

- Responsibilities:
  1. to provide a solution for the user requirements 1, 2, 4, 5 and 6 (see Section 3.2.1).
  2. to provide list of agents which can be edited by the user.
  3. to provide a GUI to allow the user to communicate with any of those agents.
  4. to initialize the Agent Watcher upon user request.
- Permissions: full access to the Queen agent internal information.
- Activities: whenever the user requests so...
  - load/save a project.
  - add/remove an agent from the agent list.
  - update the current information about one or more agents.
  - upload the current information to the agent being viewed.
  - for one or more agents to switch to runtime mode.
  - for all agents to perform a single cycle.
  - to edit the properties of an agent being viewed.
- Protocols: this role interacts with:
  - the Updater (through the Queen's sendUpdateMessage() service) – using FIPA_REQUEST, as shown in Figure 4.4.
  - the Agent Watcher – no protocol used, it spawns the Agent Watcher specifying all the information about the agent to be watched.
  - the Information Retriever – no protocol used, it creates an instance of the Information Retriever every time the user requests the GUI to update from one or more agents.
  - the Subscriber – no protocol used, it creates an instance of the Subscriber whenever the user request for an agent to be watched by the Agent Watcher, and sends a stop() signal to this instance when the Agent Watcher is closed.
  - the Retained Mode Handler – sending Q_STEP and Q_RUN commands, using the FIPA_PROPOSE protocol, as shown in Figure 4.4.

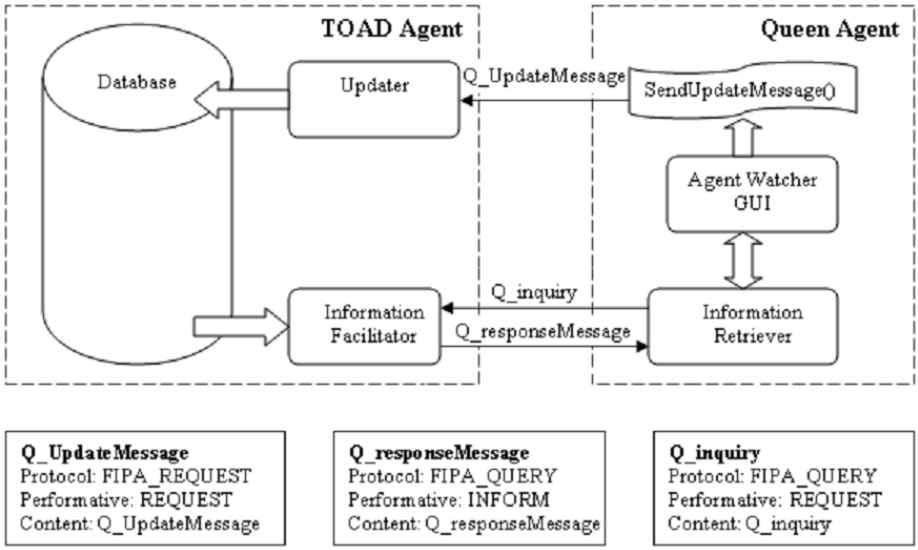


Figure 4.5: Agent Watcher GUI and related roles.

Agent Watcher GUI

This role is instantiated every time the user wants to watch a specific agent.

- Responsibilities:
  1. to keep an up to date visual representation of a TOAD agent which is said to be watched by the Agent Watcher.
  2. to provide a solution for the user requirements 2, 3, 5, and 7 (see Section 3.2.1).
  3. to provide a GUI to allow the user to communicate with the agent being watched.
  4. to make sure the channel inputs and channel outputs of the various agents are consistent, i.e. if the user adds the agent "Frog" to the valid outputs of the agent "Toad", then it is the Agent Watcher's responsability to automatically add the agent "Toad" to the valid inputs of the agent "Frog".
- Permissions: it should only have access to the AgentSpec object with which it has been initialised and to the sendUpdateMessage service by the Queen.
- Activities:
  - load/save an .oclp file.
  - update the current information about the agent being watched.
  - upload the current information to the agent being viewed.

  - automatically update related agents valid input when valid outputs are updated (when the user requests the GUI to upload the information and when the user removes an entry from the valid outputs.

- Protocols: this role interacts with:
  - the Updater (through the Queen's sendUpdateMessage() service) − using FIPA_REQUEST. See Figure 4.5.
  - the Information Retriever − no protocol used, it creates an instance of the Information Retriever every time the user request an information download. See Figure 4.5.
  - the Subscriber − no protocol, the interaction is passive. When created, the subscriber will have a reference to the Agent Watcher so as to keep it updated. See Figure 4.6.

Information Retriever

This role is instantiated whenever up to date information from an agent is required.

- Responsibilities:
  1. to retrieve information from an agent.
  2. to update the relevant GUI interfaces when the process is completed.
  3. to update the internal Queen database.
- Permissions: access to the Queen's extractQ_responseMessage() service as well as full access to the relevant fields of the Main GUI interface and Agent Watcher interface (if appropriate), components displaying: agent name, OCLP program, semantics specification, current cycle, valid inputs, valid outputs, agent status and last output produced.
- Activities:
  - to send the information query.
  - to interpret the response by the agent, and update the fields specified above.
- Protocols: this role interacts with:
  - the Information Facilitator − using the FIPA_QUERY protocol, as shown in Figures 4.4 and 4.5.
  - the Main GUI − no protocol used, it updates the GUI when the information from the agent has been retrieved. See Figure 4.4.
  - the Agent Watcher (if the agent the role is communicating with is being watched) − no protocol used, it updates the GUI when the information from the agent has been retrieved. See Figure 4.5.

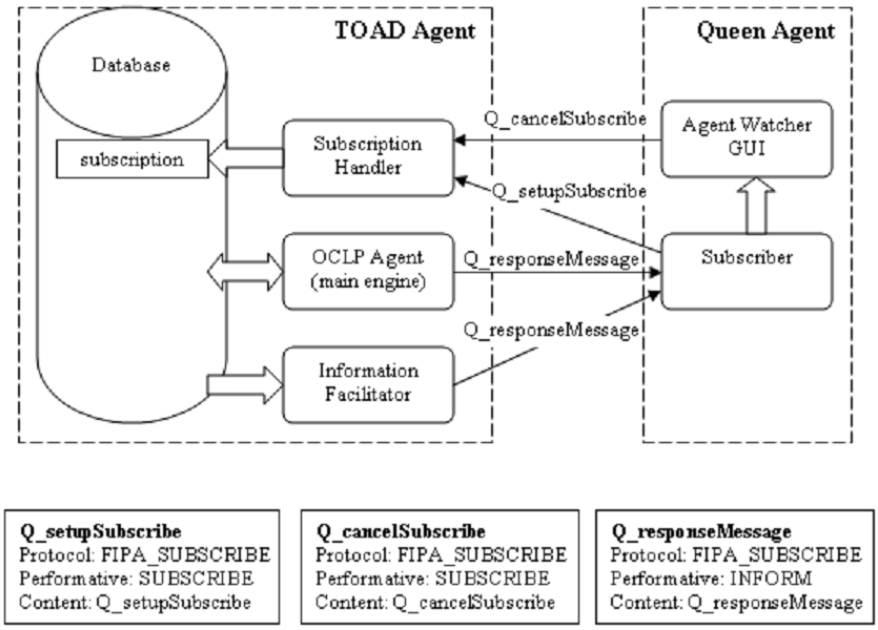


Figure 4.6: Subscriber and related roles.

Subscriber

- Responsibilities: to keep the Agent Watcher's GUI up to date.
- Permissions: write access to the GUI components in the Agent Watcher.
- Activities:
  - to initiate the subscription sending a Q_setupSubscribe to an agent.
  - to update the GUI components in the Agent Watcher when a message from a subscribed agent is received.
- Protocols: this role interacts with:
  - the Subscription Handler − using the FIPA_SUBSCRIBE protocol, as shown in Figure 4.6.
  - the Agent Watcher − no protocol used, it actively updates its fields. See Figures 4.5 and 4.6.

4.3.5 Behavioural implementation

It is easy to see JADE behaviours as roles, because although they all run in the same execution thread, they are essentially autonomous and tend to interact through

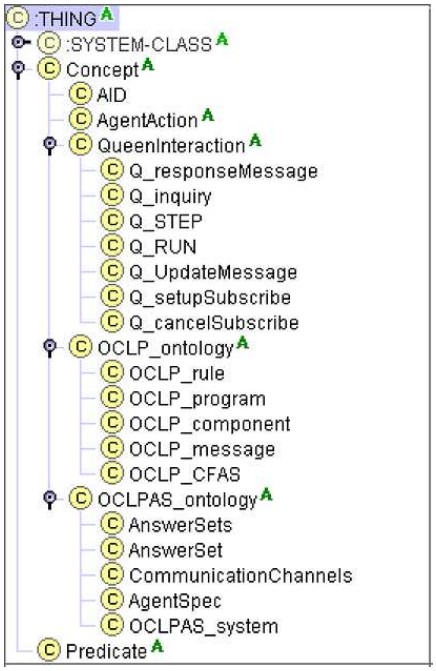


Figure 4.7: TOADs ontology − tree structure (extract from Protégé).

protocols. Here we must deviate from the GAIA perspective as we ought to define the interactions between roles that are assumed by the same agent. Hence we shall distinguish between internal interactions (within an agent), for which we describe an interface, and external interactions (between two or more agents), for which we design the FIPA compliant protocol. Next chapter presents the actual implementation of their internal processes.

4.4 TOADs ontology

An ontology is a grounding for a set of concepts and relations between them, it gives a way for interacting agents to be unambiguous about the meaning of a given object.

In TOADs, the ontology is simply a hierarchic class structure, produced by the beangenerator plug-in [Aart et al 2002.] under Protégé. Following the requirement specification and the architecture of the domain to be described the following ontology was designed. See Figure 4.7.

Q_responseMessage

Used to send the queen information about the agent.

- Boolean fixpoint − whether a fixpoint has been achieved.

- CommunicationChannels commChannels.
- Boolean credulous − true for credulous semantics, false for sceptic semantics
- OCLP_program internalOCLP.
- int cycle − the current cycle the agent is running.
- Boolean evolutionaryFlag − whether the last two cycles were essentially identical (same input and same output).
- AnswerSetslastOutput − the latest answer sets calculated.

Q_inquiry

Commands an agent to retrieve information to the Queen.

Q_STEP

Commands an agent to make a single step (initiate the Main Engine and issue the Calls for Answer Sets).

Q_RUN

Commands an agent to run in runtime mode.

Q_UpdateMessage

- AID toAgent − destinatary.
- Boolean doOCLP − whether the agent should update to the newOCLP.
- Boolean doSemantics − whether the agent should update to the newSemantics.
- Boolean doCommChannels − whether the agent should overwrite his communication channels to the ones in newCommChannels.
- Boolean removeChannels − whether the agent should remove from his channels the ones in newCommChannels
- Boolean appendChannels − whether the agent should append to his channels the ones in newCommChannels.
- OCLP_program newOCLP.
- Boolean newSemantics − true for credulous, false for sceptic.
- CommunicationChannels commChannels

Q_setupSubscribe

Commands an agent to start sending information to the queen when relevant events occur, under a special subscription protocol. These will be sent when the agent produces a new Answer Set or when the Queen request information. It shall be used to keep the Agent Watcher up to date.

Q_cancelSubscribe

Commands an agent to stop sending the regular information, whose process was started by a Q_setupSubscribe message.

OCLP_rule

- List head − list of atoms (Strings) in the head of the rule. Assumed to be XOR'd.
- List body − list of atoms (Strings) in the body of the rule. Assumed to be in conjunction.

OCLP_program

- List components − a list of OCLP_component objects. The components of the OCLP program.
- List relations − a list of Strings of the form "xxxxx ¡ yyyyy" where xxxxx and yyyyy are component names.

OCLP_component

- List rules − a list of OCLP_rule objects. The rules of the component.
- String name − the name of the component.

OCLP_message

- AnswerSets answerSets − the answer sets to be communicated.
- :THING auxiliary − unused by TOADs. For the user to implement.
- int cycle − the current cycle. Used to ensure synchrony.

AnswerSets

- List list − a list of AnswerSet objects. The collection of answer sets.

AnswerSet

- List positive − list of atoms (Strings) belonging to the positive part.
- List negative − list of atoms (Strings) belonging to the negative part.

CommunicationChannels

- List channelInput − a list of AID's. The valid input, i.e. the only agents from which the agent will expect and accept OCLP_message messages.
- List channelOuptut − a list of AID's. The valid output, i.e. the agents to which the agent will send her OCLP_message message at the end of every cycle.

AgentSpec

A data structure required by the Main GUI to keep track of the agents in the list.

- AID id − the agent's AID.
- List channelInput − as in CommunicationChannels.
- List channelOutput − as in CommunicationChannels.
- Boolean isBeingMonitored − whether or not there is an Agent Watcher monitoring this agent.
- String nickname − the agent's short name for local agents, the agent's full name for agents in other platforms.
- AnswerSets lastOutput − the last output answer sets the agent calculated.

OCLPAS_system

The internal database containing all the information about the system.

- List agentsSpecification − a list of AgentSpec objects, one for each agent in the Queen's agent list.

4.5 Synchrony and the CFAS protocol

TOADs agents need a way of knowing when their valid outputs are ready to process their input, this is the motivation for our CFAS (Call For Answer Sets) protocol, which controls the lifecycle of TOADs agents. CFAS messages are sent with a given deadline, after which the TOAD agent will carry on using the empty set as input for those agents which did not reply in time. Once all input has been received the solution to the updated OCLP is computed immediately and evolutionary status is

revised. After this, the agent will wait for another time lapse before declaring the cycle completed, giving time for slower agents to send their CFAS requests.

Once the cycle is declared completed, runtime mode agents will start the next cycle while retained mode agents will wait for the Queen's command to proceed.

4.5.1 Input agents and Starting agents

As described in Chapter 2 page 23, an OCLPAS has a special couple of agent types: the **input agent** and the **output agent**; the first has no valid inputs and the latter has no valid outputs. An input agent will not send any CFAS messages and its program will return the same output always, hence it suffices to calculate the final Answer Sets for this agent once. The CFASListenerBehaviour (described below) shall take care of the rest. For the same reason, this agent will reply to CFAS for any cycle indistinctively.

Another special kind of agents are those declared as **starting agents**. Cyclic[1] OCLPAS need a way to start the cascade effect for every cycle. Without a clear distinction all agents would issue their CFAS for cycle 1 and never obtain an answer. Therefore we allow for a special kind of agents called **starting agents** which will reply to CFAS for one cycle ahead. See Chapter 6 for details on how to declare a starting agent.

4.5.2 CFAS timers

In order to ensure robustness, requirements state that a timeout mechanism must be designed. In TOADs we specify two timers: the CFAS timer and the OCLP_message timer. The first is the amount of time an agent will wait for CFAS to come once the current cycle is completed; the latter is the amount of time an agent will wait for her CFAS to be replied.

This way, CFAS are ensured to be answered as soon as possible, avoiding parts of the system to delay others.

4.6 Towards an implementation

As mentioned in Section 4.3.5, we shall implement each one of the roles described as a JADE behaviour; in the next chapter, a concrete view of these behaviours is given, together with an analysis of events, flow control and data interchange among them.

[1] OCLPAS where all agents have at least one valid input

Chapter 5

Implementation

In this chapter, each role (as seen in Chapter 4) is implemented as one or more JADE behaviours or as a swing component (in the case of the GUI roles). Pseudocode for each one of these is given; see the legend for implementation diagrams in Figure 5.1.

5.1 Technology

The system is entirely written in Java Standard Edition 1.4 using JADE 3.0b1. The graphical interfaces (files: `Queen.java, AddAgent.java, AgentEdit.java, FileDialog.java, SaveDialog.java`) were produced using BX for Java ©.

5.2 The OCLP Agent – behaviours implementation

When initialised, the OCLP agent spawns the basic behaviours and some additional behaviours to interact with the Queen, if running in retained mode, as shown in Figure 5.2.

5.2.1 The OCLP Agent role

In order to implement a JADE timeout for CFAS messages, this role ought to be split into two behaviours: a **CFASListenerBehaviour** behaviour, which will listen for incoming CFAS messages and a **MainEngineBehaviour** behaviour, which will listen for incoming OCLP_message messages (replies to the CFAS sent).

The MainEngineBehaviour behaviour shall initiate the conversation by sending a CFAS (call for answer sets), it will then wait for replies, all this by extending the AchieveREInitiator class. A CFAS is a message carrying a CFAS object specifying what cycle's AnswerSets we are requiring, using the protocol FIPA_QUERY and performative REQUEST_WHENEVER.

On creation it shall send the CFAS.

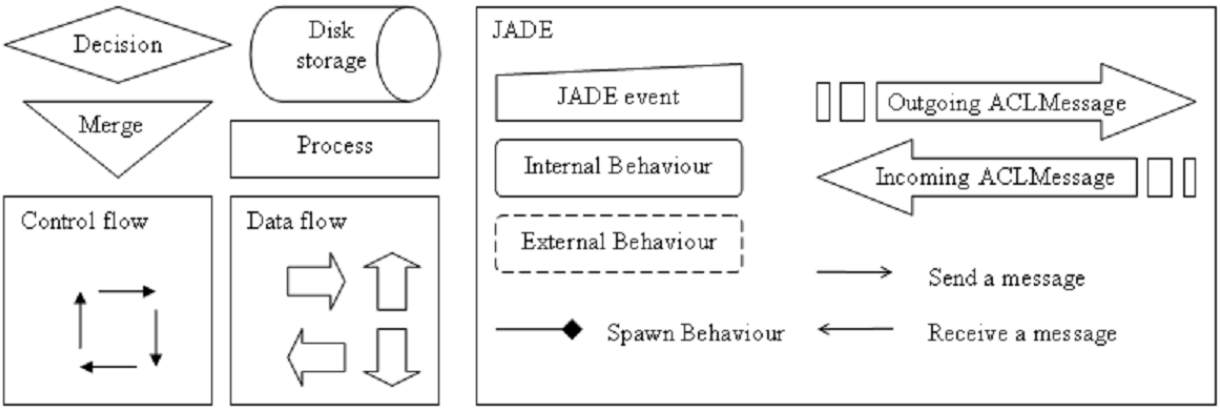


Figure 5.1: Legend for implementation diagrams

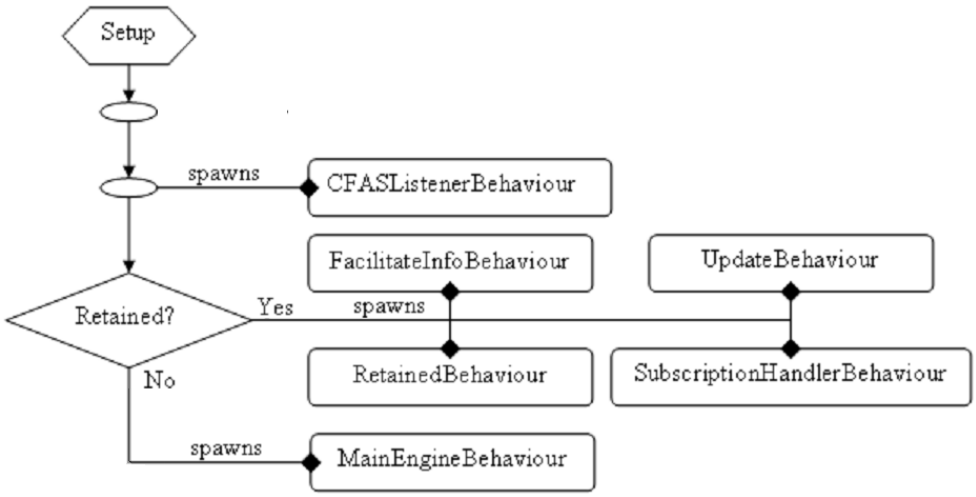


Figure 5.2: OCLP Agent: setup process.

Upon receiving a response (overriding handleInform()), the MainEngineBehaviour will add the first of the incoming Answer Sets to the merged input Answer Set.

Upon receiving all responses, or when the timeout is expired (overriding handleAllResponses()), the behaviour shall:

1. Filter the merged input, removing contradictory information; i.e. if both 'a' and 'not a' are present in the answer set, then remove both.
2. Create an updated OCLP program by extending the internal OCLP of the agent, adding an extra component with the incoming information. The component shall contain rules affirming all positive atoms in the filtered input and constraints negating atoms in its negative part. Furthermore relation rules will be added to make the new component less preferred than each other component in the program.
3. Produce a syntactically correct String corresponding to the updated OCLP_program object and feed it into an external process in which OCT is invoked.
4. Parse the output stream of the process to retrieve the answer sets.
5. Store this AnswerSets object.
6. Initiate a WakerBehaviour (a behaviour which performs an action after a certain time) to allow for some time (specified in the agent field OCLP_message_TIMEOUT) to pass before completing the current cycle. This gives a chance to slower agents to send their CFAS, when all CFAS have been received and replied, this WakerBehaviour is destroyed and the cycle is considered complete anyway. Completing the current cycle means that: in retained mode the agent will stop and wait for the next Q_STEP command from the queen whereas in runtime mode a new MainEngineBehaviour is immediately spawned.

The internal processes of the MainEngineBehaviour behaviour are displayed in Figure 5.3.

The CFASListenerBehaviour behaviour reacts to incoming CFAS messages, replying them immediately if possible, or storing them otherwise. Particularly, when all CFAS have been replied, this behaviour will be responsible for starting the new cycle (if running in runtime mode) or declaring the current cycle completed so that the agent will obey incoming Q_STEP commands, if in retained mode. This behaviour is clearly depicted in Figure 5.4.

5.2.2 The Information Facilitator role

This role shall be implemented as an AchieveREResponder, named **FacilitateInfoBehaviour**, listening for incoming messages matching the protocol FIPA_QUERY and performative REQUEST. It will simply access the internal information of the agent and retrieve it to the inquirer. Pseudocode in Figure 5.5.

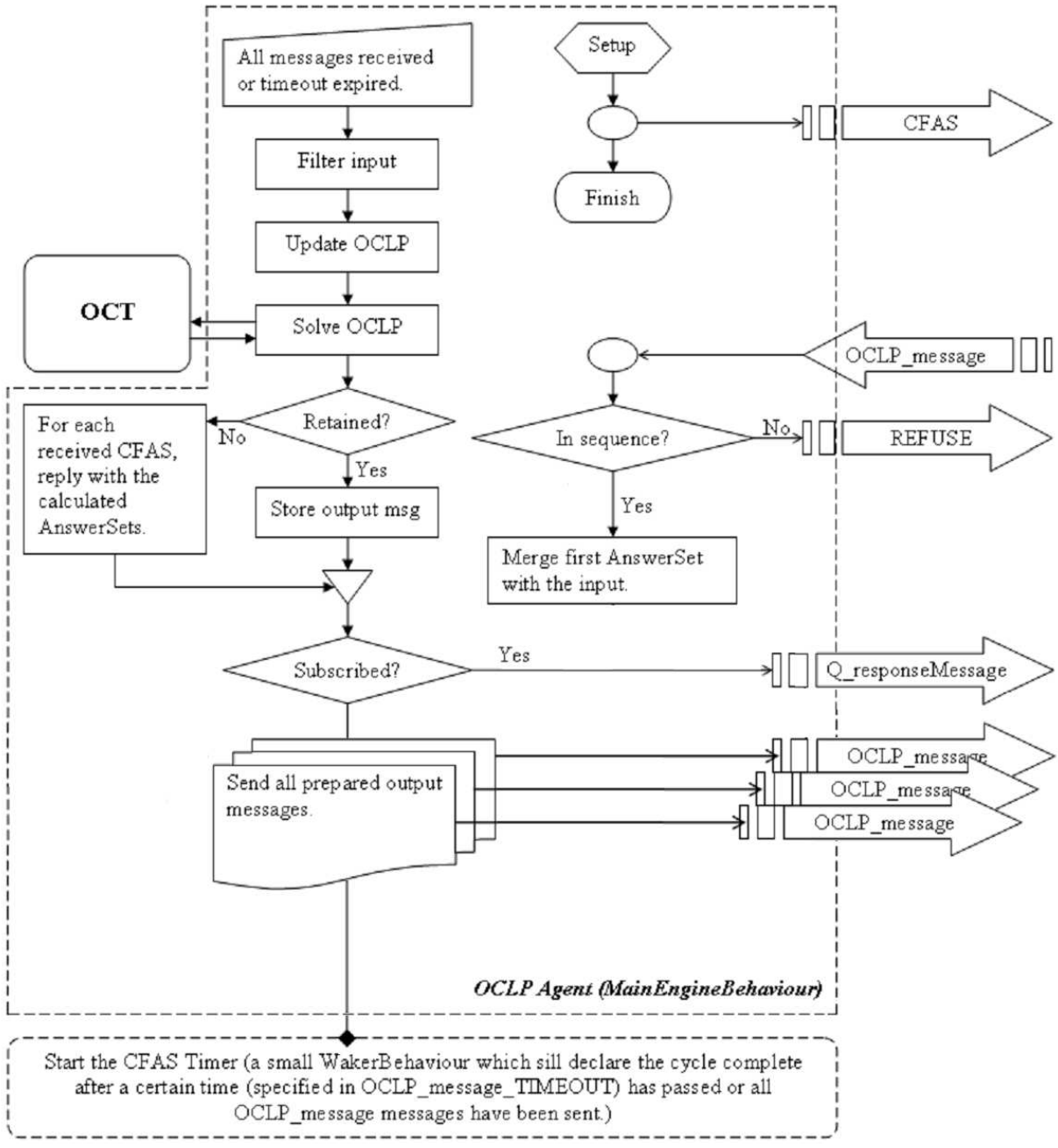


Figure 5.3: Pseudocode: MainEngineBehaviour.

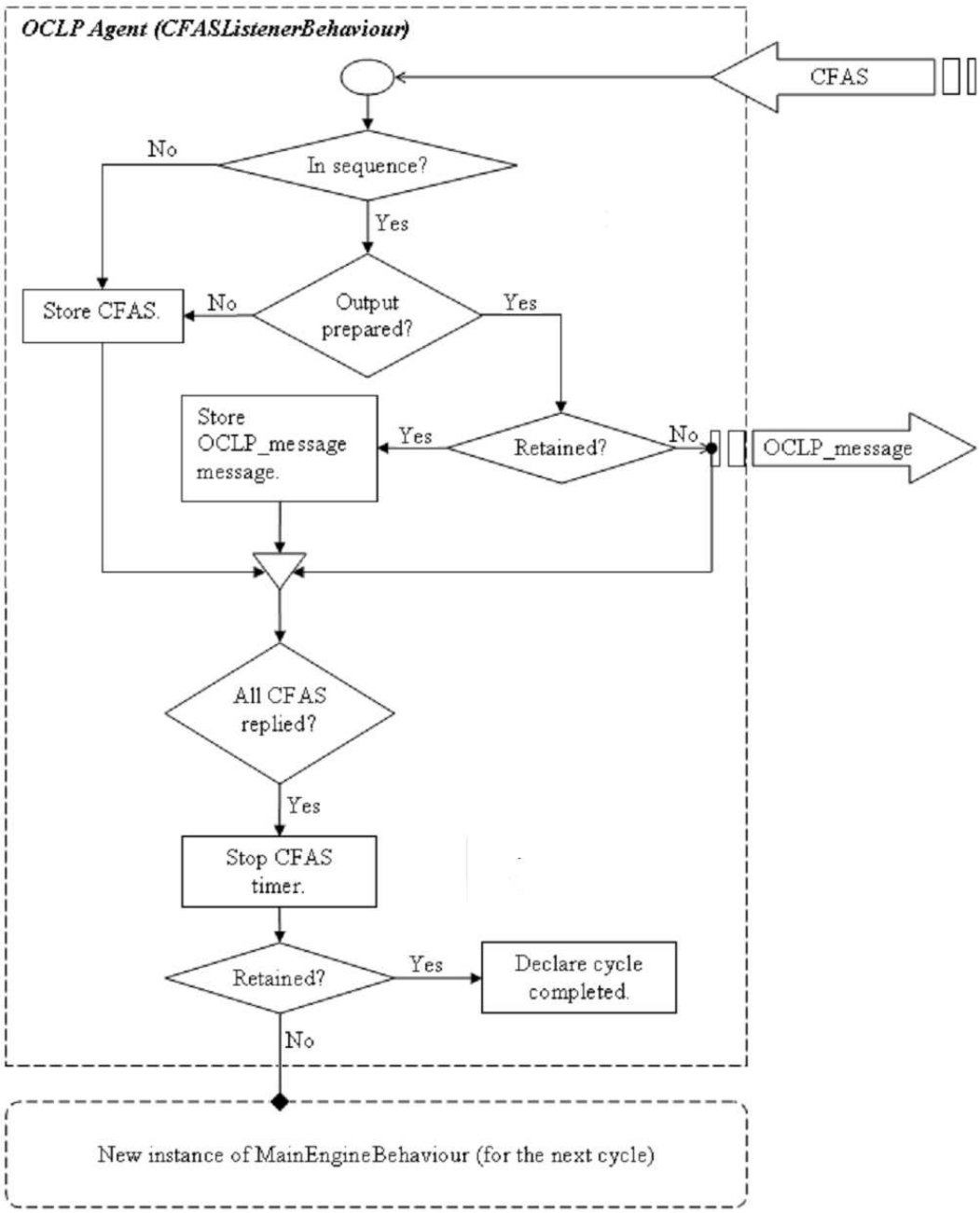


Figure 5.4: Pseudocode: OCLPAgent.CFASListenerBehaviour.

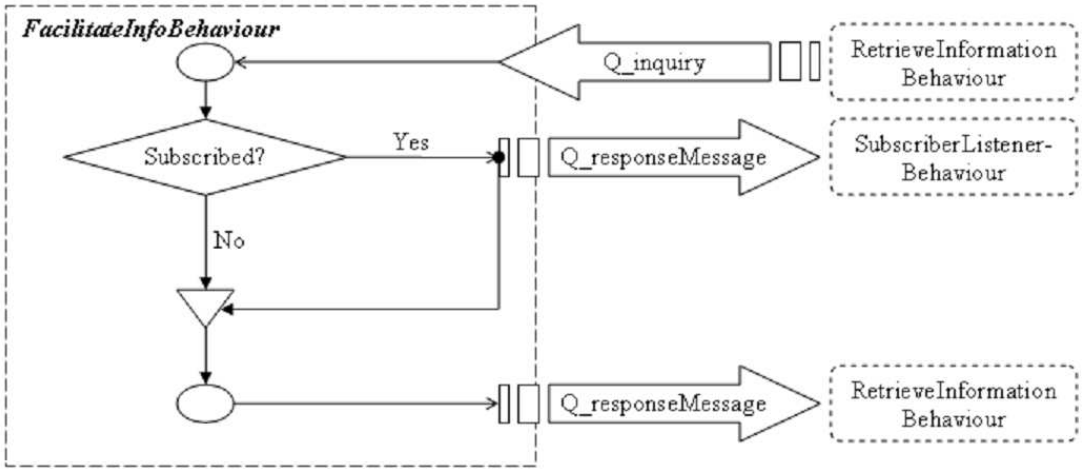


Figure 5.5: Pseudocode: FacilitateInfoBehaviour.

5.2.3 The Updater role

This role shall be implemented as a SimpleAchieveREResponder, named **Update-Behaviour**, listening for incoming messages matching the protocol FIPA_REQUEST and performative REQUEST. It will update the internal information of the agent, according to the embedded Q_UpdateMessage object. If this object is invalid, or if an error occurs, the behaviour will reply with a FAILURE message. Pseudocode in Figure 5.6.

5.2.4 The Retained Mode Handler role

This role shall be implemented as an AchieveREResponder, named **RetainedBehaviour**, listening for incoming messages matching the protocol FIPA_PROPOSE and performative PROPOSE. Inside incoming messages of this type, the behaviour will expect one of two objects: Q_STEP or Q_RUN. If none is found, or if an error occurs, the behaviour will reply with a FAILURE message. If a Q_RUN command is received, the agent will switch to runtime mode.

If a Q_STEP command is received, behaviour will check that the previous cycle is completed; if it is not it will reply with a FAILURE message, if it is a new cycle is started: a new instance of MainEngineBehaviour is spawned. Pseudocode in Figure 5.7.

5.2.5 The Subscription Handler role

This role shall be implemented as an CyclicBehaviour, named **SubscriptionHandlerBehaviour**, listening for incoming messages matching the protocol FIPA_SUBSCRIBE

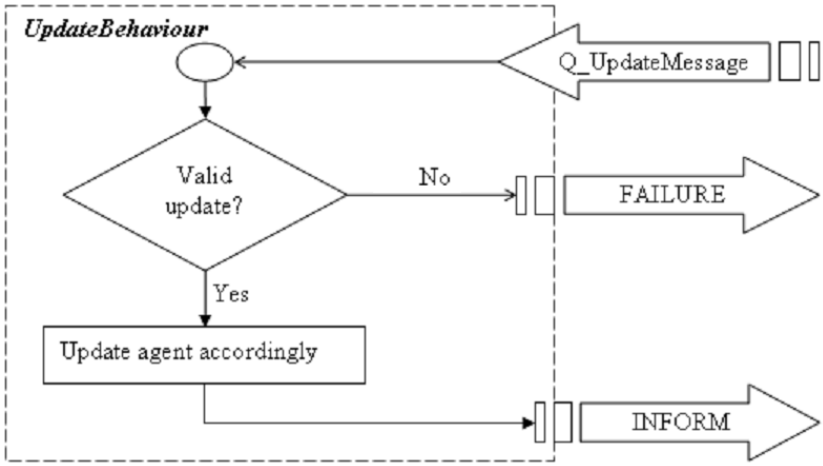


Figure 5.6: Pseudocode: UpdateBehaviour.

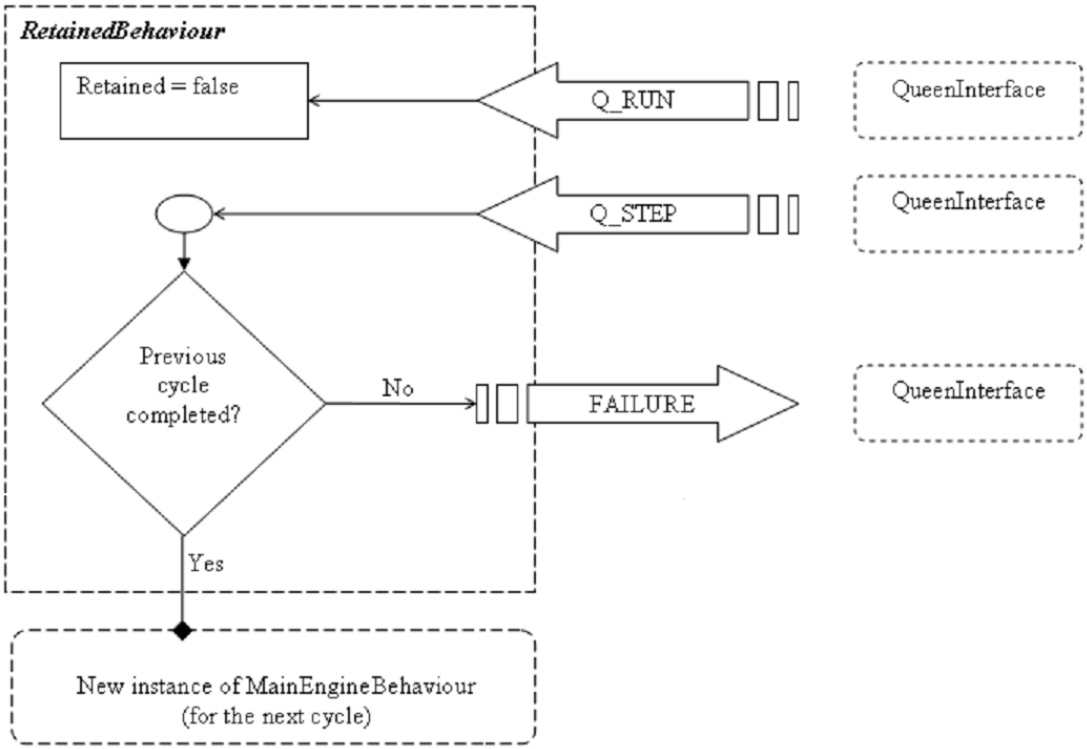


Figure 5.7: Pseudocode: UpdateBehaviour.

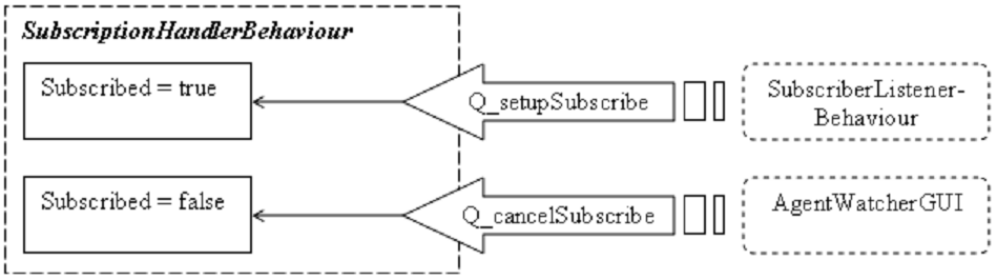


Figure 5.8: Pseudocode: UpdateBehaviour.

and performative SUBSCRIBE. Inside incoming messages of this type, the behaviour will expect one of two objects: Q_setupSubsribe or Q_cancelSubscribe. If none is found, or if an error occurs, the behaviour will reply with a FAILURE message. Intuitively according to the command received, the subscription flag will be switched on and off. Pseudocode in Figure 5.8.

5.3 Queen Agent – behaviours implementation

When initialised, the Queen agent will simply start up the user interface. All other activities are reactive to user input.

5.3.1 Interface implementation

No HCI guidelines were followed in the development of TOADs graphical interfaces; every component has been built in two layers, a graphical layer (Queen.java, AgentEdit.java, AddAgent.java) and an operational layer (to be discussed in this chapter). Users interested in enhancing the look and feel of the system must adapt the graphical layer accordingly, preserving object names. Screen samples of the main GUI and the agent watcher can be found in Chapter 6.

5.3.2 The Main GUI role

This role shall be implemented under the name of **QueenInterface** as extending a JApplet swing component and implementing an ActionListener. Therefore, it will act upon user request, as shown in Figure 5.9. This role shall also include a SendStepBehaviour behaviour which will be used to send Q_RUN and Q_STEP messages. All messages are sent to those agents selected in the JTree component.

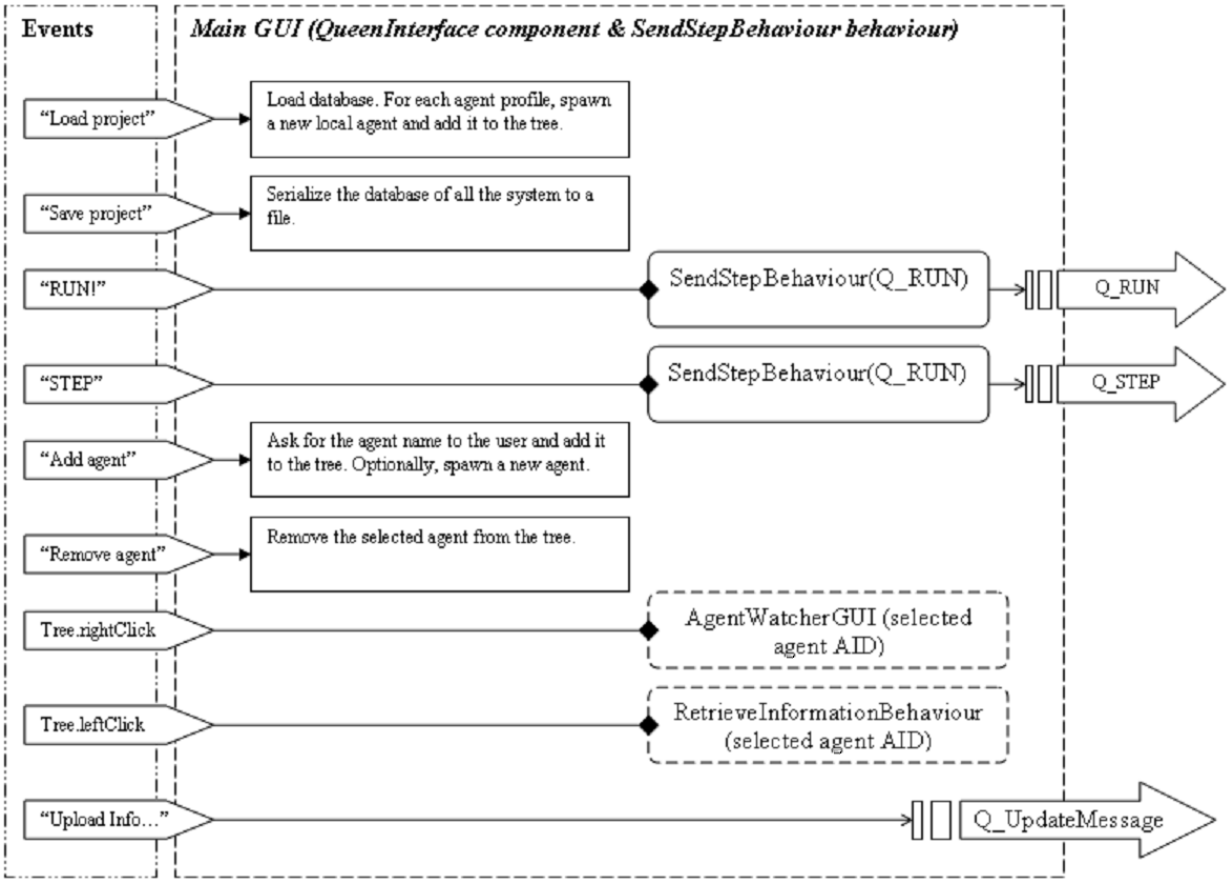


Figure 5.9: Pseudocode: UpdateBehaviour.

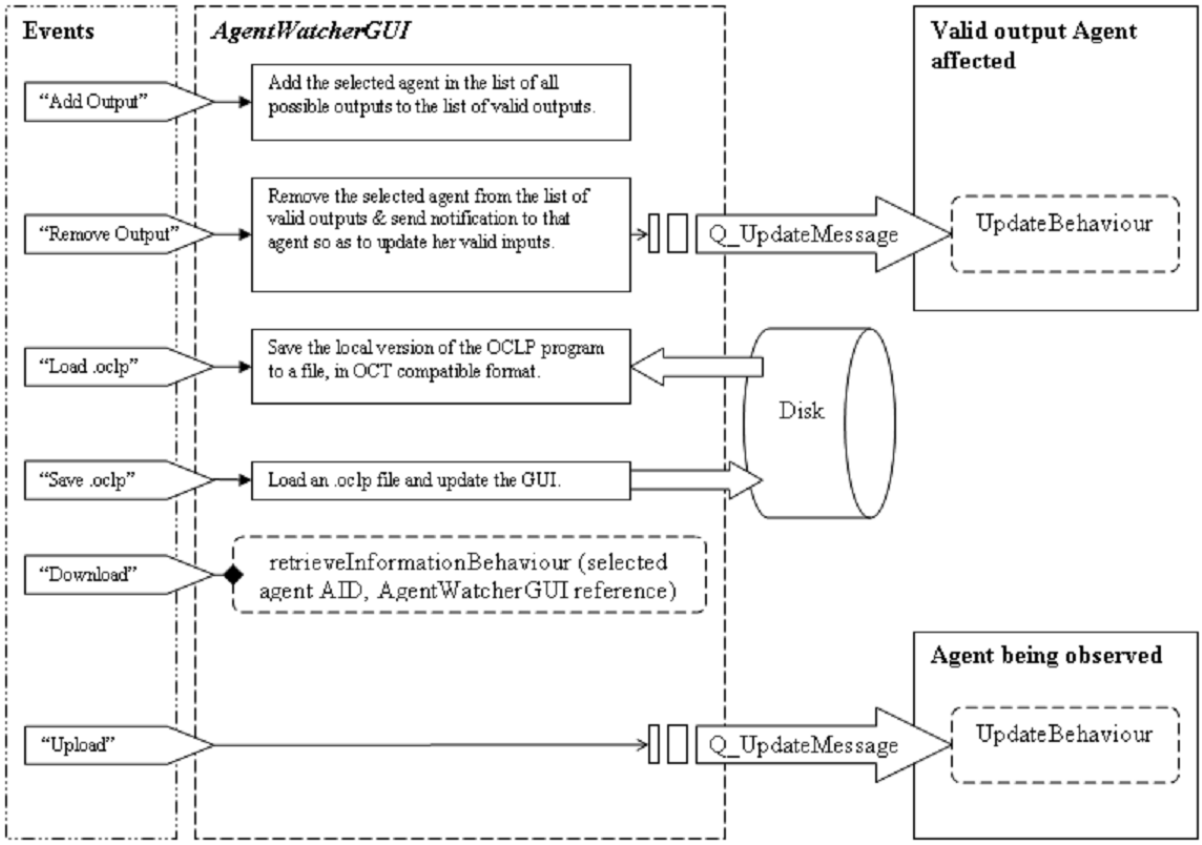


Figure 5.10: Pseudocode: UpdateBehaviour.

5.3.3 The Agent Watcher role

This role will be written as another swing component, **AgentWatcherGUI**, extending JDialog and implementing ActionListener, dependant on the Main GUI Applet. It will be initiated by the main GUI with a reference to the agent to oversee. The detailed functionality of this role is shown in Figure 5.10.

5.3.4 The Information Retriever

This role shall be implemented as a single JADE behaviour, extending SimpleAchieveREInitiator, called **RetrieveInformationBehaviour**. Both the Main GUI and the Agent Watcher may spawn this behaviour to retrieve information from an agent. If created by an Agent Watcher, an extra reference must be given so that once the information is obtained, the behaviour can actively update the GUI components. See pseudocode in Figure 5.11.

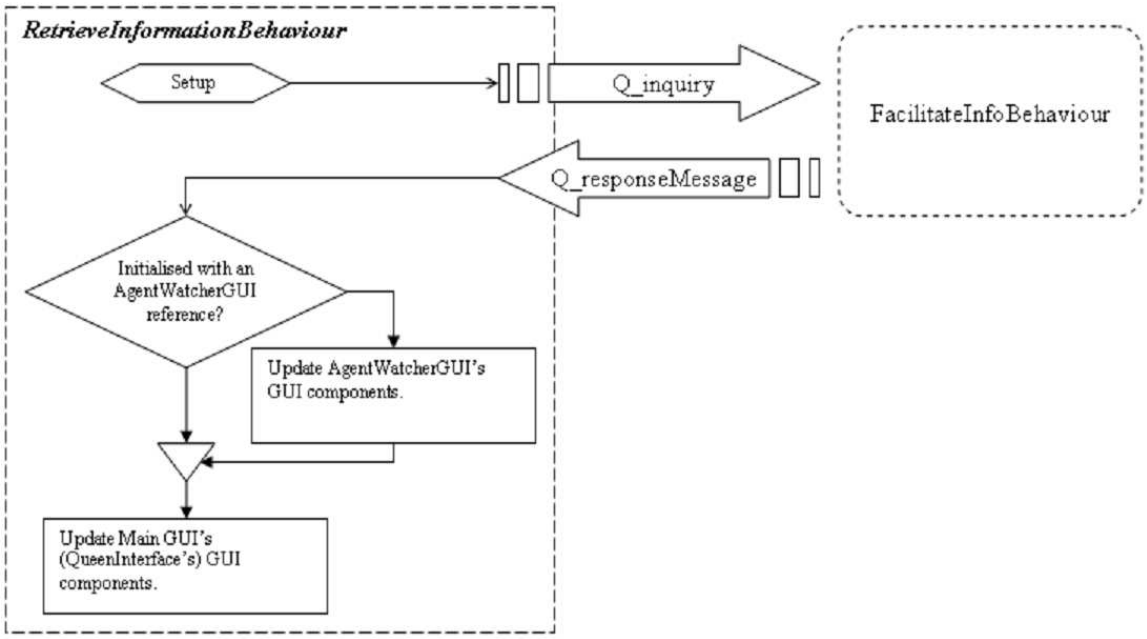


Figure 5.11: Pseudocode: RetrieveInformationBehaviour.

5.3.5 The Subscriber role

This is a SimpleBehaviour behaviour, called **SubscriberListenerBehaviour**, listening for messages matching the protocol FIPA_SUBSCRIBE and performative INFORM. It will be initiated at the same time the Agent Watcher and it will be given a reference to the agent to subscribe. When initiated it shall send a Q_setupSubscribe message to the specified agent. This process is shown in Figure 5.12.

5.4 Testing

The testing of TOADs was carried out in a white-box fashion: dummy agents (not included) were used to verify responsiveness from behaviours and dummy programs (not included) were used to text the auxiliary services.

Early stages of the project involved attempting to implement a direct communication between agents (i.e. agents would send their outputs as soon as they are ready), testing this method proved unfruitful as part of the system (which were artificially altered to simulate delay) would not be able to handle incoming information. This is way the CFAS protocol was designed.

Once implemented, the CFAS protocol displayed a similar behaviour to the first version of the program: this time delays were retroactively imposed on consequent agents and information was lost. At this point the protocol was reformulated to accept

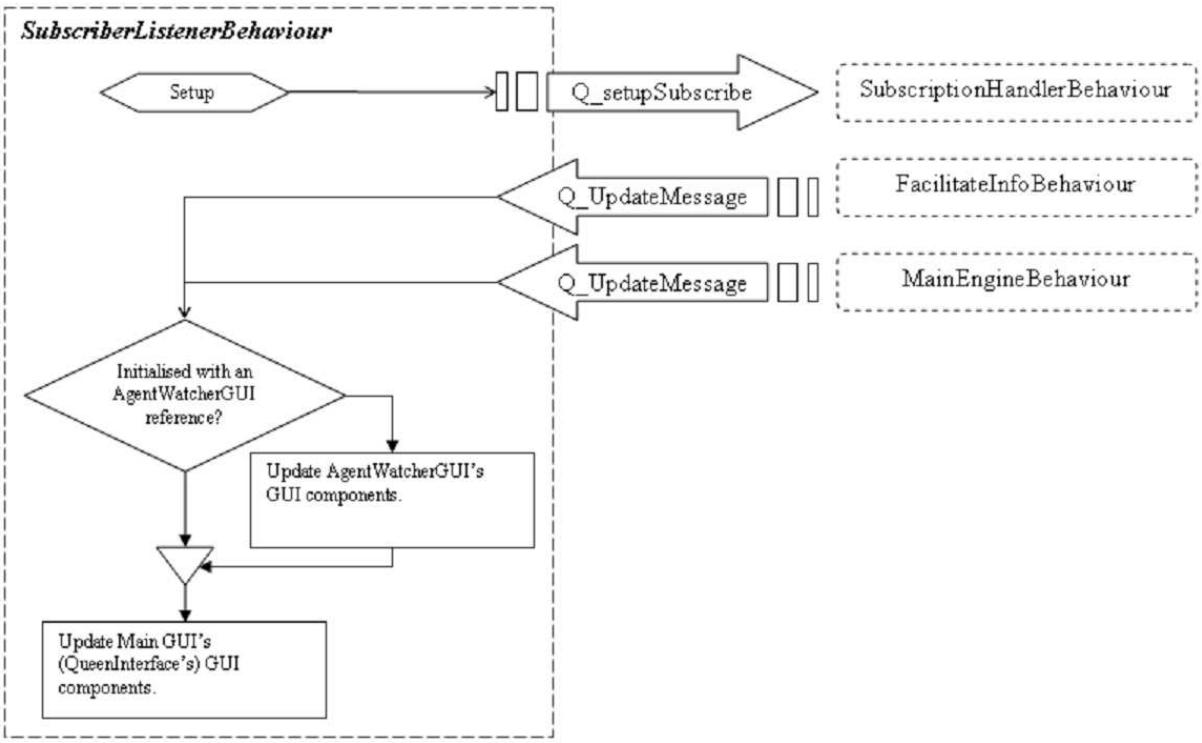


Figure 5.12: Pseudocode: UpdateBehaviour.

CFAS for future cycles and storing them so as to respond whenever the answer for that cycle became ready.

Nonetheless, the cascade effect of the delay was still present and agents which did not receive answers in time would in turn not provide their answers in time. In order to solve this inconvenient, two separate timers were implemented: one defining the time to wait for CFAS to be answered and one defining the time to wait for incoming CFAS once a cycle is completed, before starting the next cycle.

It is important to highlight that these timers do not solve the problem on their own, and users should pay special attention to the timeout assignation so that the time agent A waits for a reply to a CFAS has to be safely[1] larger than the time her valid inputs wait for replies to their CFAS messages.

During the final phase of the testing of TOADs, while studying its relation to Game Theory, a major flaw was discovered in the CFAS protocol: system which did not have any Input agents [2], such as the OCLPAS representations of strategic games, would never start running as all agents would wait for an Input Agent to start the CFAS cascade effect[3]. At this stage the decision was made to identify a new type of agents, called **starting agents** which would run a cycle ahead as far as CFAS are concerned. Starting agents have the property that they will reply to CFAS for cycle N+1 while running cycle N.

For a demonstration of the use of TOADs, see Chapter 7.

5.5 Summary

This chapter concludes the development of TOADs. The following table summarizes other difficulties encountered during development/testing and gives a reference to the section describing its solution.

Requirement	Solution	Reference
OCLP_message message passing synchrony	JADE's AchieveRE protocol	the CFAS protocol (Figure 5.3 and 5.4)
Need to supervise some agents closely	A subscription protocol	Agent Watcher GUI (Section 4.3.4)
Cyclic OCLPAS will not start	Special **starting agents**	Chapter 4, page 55

[1] Adjust according to hardware and computational cost of the agents.

[2] Input agents are gents without valid inputs, which always produce the same output and hence respond to any CFAS immediately.

[3] CFAS for cycle N+1 are issued at cycle N, hence the whole system (running cycle 0) will be blocked unless an input agent replies to a CFAS for cycle 1, and when it does, agents will recursively reply to their CFAS.

In the next chapter, we take a look at the final product and its usage in both interactive and modular modes.

Chapter 6

How to use TOADs

6.1 Installation and System Requirements

TOADs comes as a single file, `toads.jar`, which has to be included in the CLASSPATH of the operative system. TOADs is guaranteed to work in Unix/Linux. In order for TOADs to run, OCT and JADE must be installed and the beangenerator plugin must be in the CLASSPATH as well.

6.2 Initialising a TOAD agent

From now on, assume the file `toads.jar` is in the classpath and so are the OCT, JADE and beangenerator files. From the shell commandline, type:

```
java jade.Boot -container worldpeace.toads.OCLPAgent
```

or

```
java jade.Boot -container "worldpeace.toads.OCLPAgent(options)"
```

`options`: one or more of the following, separated by commas:

queen:<Queen name>	the name of the Queen to obey. If specified the agent will start in retained mode. Otherwise it will start in runtime mode.
oct:<OCT path>	specifies oct command path, default is "./oct".
parameters:<parameters>	additional oct parameters.
log:<file>	activate the logging tool and write to the file specified in <file>.

open:<oclp file>	initialise with the program in <oclp file>.
semantics:[c|s]	specifies the semantics to be used by the agent. c for credulous, s for sceptic. Default is credulous.
input:<agent>	adds <agent> to the list of valid inputs.
output:<agent>	adds <agent> to the list of valid outputs.
starter:true	declares this agent a starting agent.

This will create a new JADE container and initialise the TOAD agent, as described in Chapter 5.

For example, the command line:

```
java jade.Boot -port 1234 -container "Toad:
worldpeace.toads.OCLPAgent(queen:Queen,oct:~/oct/oct,open:ToadBrain.oclp,
semantics:s,input:Frog,input:Tadpole,output:Croak@WORLDPEACE:1234/JADE)"
```

will initialise a new container and an agent called `Toad` in the local JADE platform using the port 1234. The agent will start in retained mode and listening for commands coming from the queen agent `Queen`. Its internal program will initially be loaded from the file `ToadBrain.oclp` and the agent will run using sceptic semantics. It shall expect messages from local agents called `Frog` and `Tadpole` and it will send its output to an agent called `Croak`, which is running on the remote platform WORLDPEACE:1234/JADE. We assume that oct is in the ~/oct folder.

6.3 Initialising a Queen Agent

From the command shell, type:

```
java jade.Boot -container Queen:worldpeace.toads.QueenAgent
```

This will start the Queen agent and the graphical interface. Once started the main GUI should appear. Figure 6.1 shows the GUI and its commands. Notice that double-right-clicking on the agent list will bring up an Agent Watcher window, as the one shown in Figure 6.2.

Figure 6.2 displays the Queen agent GUI; following, a summary of the various components:

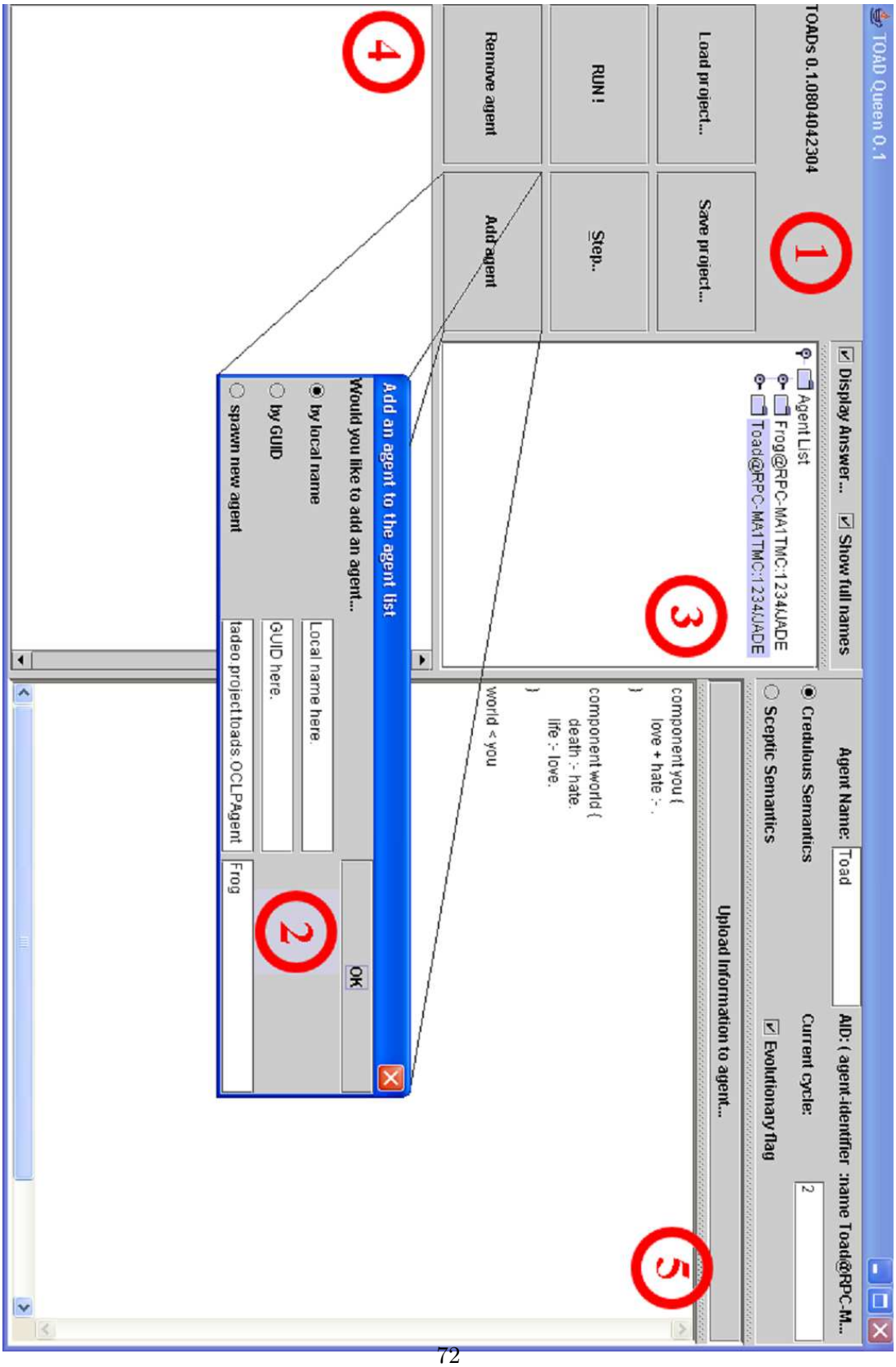


Figure 6.1: The Queen agent, graphical interface.

1. The main panel, displays version and gives access to system-specific functions, of self-explanatory functionality. Notice that "RUN!" and "STEP" will send a Q_STEP/Q_RUN command to all agents.

2. Add an agent to the agent list. It may be an existing agent, alternatively a new agent can be spawned in the Queen's platform (not recommended).

3. Agent list: clicking an agent will retrieve information from it; double-right-clicking on an agent will initiate the Agent Watcher.

4. The console, the queen standard output will be redirected here. (Not implemented)

5. Display/Edit internal properties of the selected agent. In order for changes to take effect, click on "Upload information to agent".

Figure 6.3 displays the Agent Watcher GUI, whose components are:

1. Up-to-date internal OCLP of the agent (including, when appropriate, the updated OCLP with respect to incoming information from other agents).

2. The list of valid outputs.

3. The last output calculated by this agent (and sent to others).

4. The list of other agents in the system. Choose an agent from this list and click "Add output" to add it to the list of outputs.

6.4 Interactive use

Whenever the Queen agent is run and one or more TOAD agents are running in retained mode listening to that Queen, the system is said to be running in interactive mode, since the human user input will be required for the progress of the system, or, at least, part of it. This use is recommended for users who wish to follow the evolution of the system closely, for whatever purpose.

6.5 Modular use

TOADs, and in particular the OCLPAgent class, was developed to allow a programmer user to reuse this class and make a custom use of the TOAD engine. Next, we present the OCLPAgent class public interface, its relevant public fields and the event methods to be overridden by the programmer. We present also a summary including several relevant services available from the OCLPAgentTools class, as static methods.

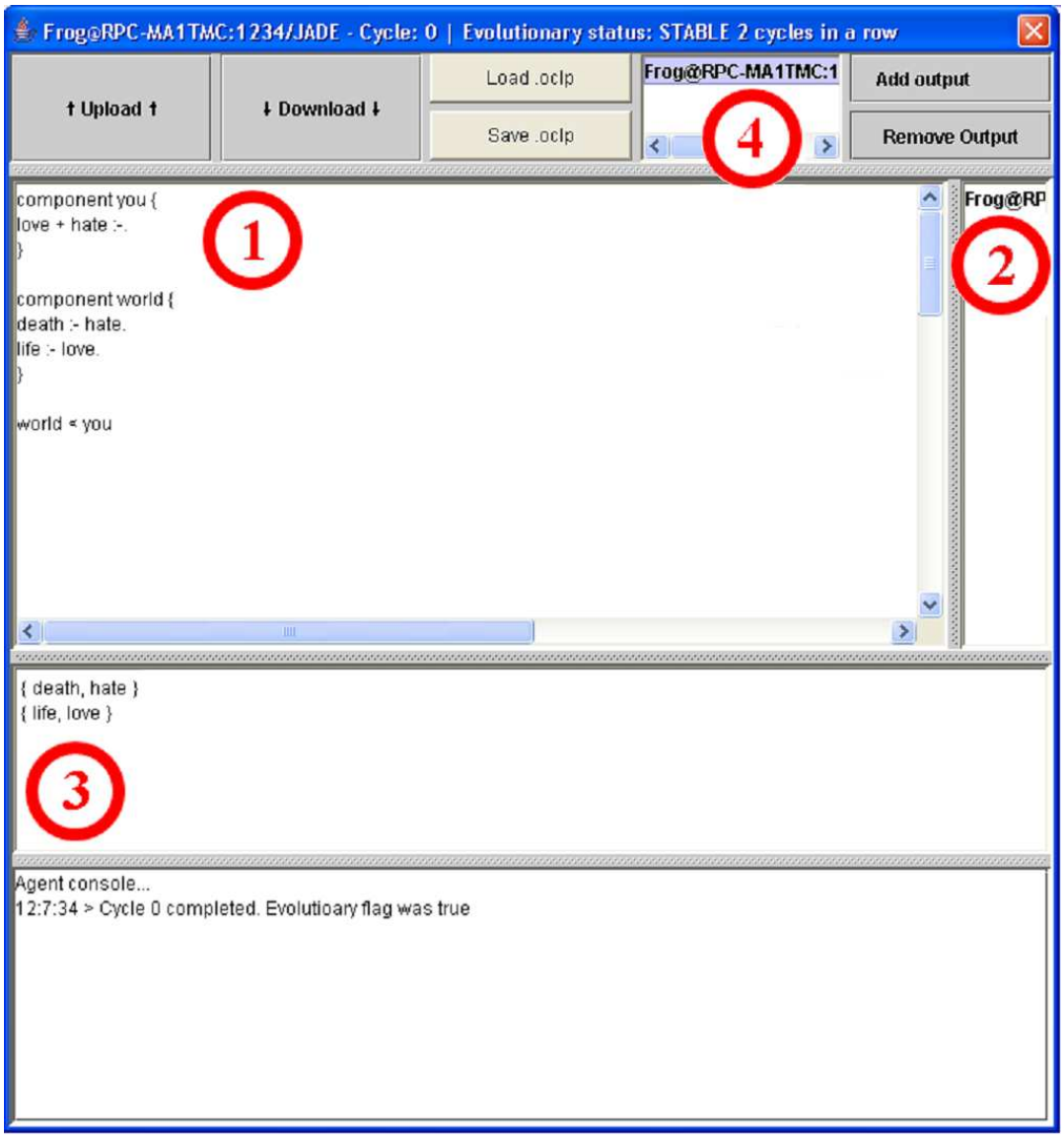


Figure 6.2: The Agent Watcher window, graphical interface.

6.5.1 Auxiliary services provided

TOADs includes a special class, **OCLPAgentTools**, intended to be reused by Java programmers; while some of its services (in the shape of static methods) need a reference to an OCLPAgent, most of them can be used in any context. Notice that due to the nature of this services, the TOADs ontology has to be used. A complete description of OCLPAgentTools services can be found in Appendix C; among them, perhaps of particular interest for the reader are:

- `public static OCLP_program upDateOCLP(OCLP_program oldProgram, AnswerSet AtomSet, String identity)`
  // update the given OCLP_program using the given set of atoms and name the new component "toads"+identity
- `public static OCLP_program OCLP_program_fromString(String program)`
  // An OCT format compatible OCLP program parser
- `public static boolean AnswerSets_compare(AnswerSets ref_list1, AnswerSets ref_list2)`
  // Compare the given AnswerSets to see if they are essentially the same
- `public static AnswerSets solveOCLP(OCLP_program program, boolean credulous, String OCT_PATH, String parameters)`
  // Invoke OCT to solve the given OCLP_program, with given semantics (credulous true for credulous, sceptic otherwise), oct path and additional parameters for OCT

6.5.2 The OCLPAgent public interface

The following fields may be accessed through the standard getXxxx/setXxxx methods (for a full list see Appendix B.1):

AID TheQueen	the AID of the Queen the agent will respond to.
boolean EvolutionaryFlag	whether the last two inputs in a row are the same and the last two outputs in a row are the same.
OCLP_program OCLP	the internal OCLP program.
boolean Credulous	semantics specification, true for credulous, false for sceptic.
CommunicationChannels commChannels	the communication channels.

6.5.3 OCLPAgent event methods

The programmer should override the following methods:

- `public void TOADs_AgentStarted()`

  This event will fire when the agent is started. It is intended for the user to keep everything stopped until he/she so desires; the programmer should override this

method to delay the agent's first cycle. When the control is returned to the agent, it will initialise its behaviours.

- `public boolean TOADs_achievedEvolutionaryFixPoint(int currentcycle, AnswerSets lastOutput)`

  This event occurs when the evolutionary fixpoint has been achieved, giving the current cycle of the agent and the final answer sets. The overridden version of this method should return `true` if the programmer wishes the agent to carry on with its normal life or `false` if the programmer wishes the agent to switch to retained mode.

- `public AnswerSets TOADs_newOutputReady(int currentcycle, AnswerSets lastOutput)`

  This event occurs whenever a new output is calculated, i.e. when a cycle is completed, giving the current cycle and the new output calculated. It is expected to return the lastOutput object with whatever alteration the user wishes to perform.

- `public OCLP_CFAS TOADs_CFASReceived(int currentcycle, OCLP_CFAS cfas)`

  This event will fire whenever a CFAS message is received, the embedded OCLP_CFAS object is passed and the function is expected to return this same object with whatever alteration the user wishes to perform on it.

- `public OCLP_message TOADs_OCLPmessageReceived(int currentcycle, OCLP_message OCLPmessage)`

  As with the previous event, but for incoming OCLP_messages.

- `public void TOADs_CFAStimeoutExpired(int currentcycle)`

  This event will fire if the agent has been waiting for incoming CFAS for long enough for its timeout to expire but one of the agent's valid outputs did not send a CFAS.

Chapter 7

TOADs and Game Theory

7.1 Introduction

[De Vos & Vermeir 2001] states it is possible to represent an extensive game with perfect information as an OCLPAS where each agent corresponds to the reasoning of a player and the system to the structure of the game.

In this chapter, TOADs is exhibited at work, exemplifying its interactive[1] use (so as to obtain the Nash equilibria and subgame perfect equilibria of a simple game) and its interactive use, for demonstration purposes.

7.2 Extensive games with perfect information

An extensive game with perfect information [Osborne & Runinstein] is a sequence of decisions taken by independent players which are given a complete list of relevant events (decisions) that happened in the past. Players base their decisions on this information, seeking to maximise the payoff value of the final history, in a rational manner. Formally

Definition 7.1. *An **extensive game with perfect information** is a 4-tuple* $\langle N, H, P, (\geq_i)_{i \in N} \rangle$*, where:*

- N is the set of players.
- H is the set of histories[2].
- $P : H \rightarrow N$ is the function that tells which player's turn it is after a certain history.

[1] Files with this chapter's examples are included in TOADs.

[2] A **history** is the concatenation of consecutive players' actions, if it includes an action for every step in the game, then it is called a **terminal history**.

- $(\geq_i)_{i \in N}$ is the preference relation of each player with respect to the possible terminal histories.

Definition 7.2. *The set of decisions that a certain player* i *makes is called the* ***strategy*** *for player i. A* ***strategy profile*** *for a game is a strategy for each player.*

Definition 7.3. *Given a game G, the* ***utility function*** *for a player i,* $T_i(s) : \{s \mid$ *s is a strategy profile*$\} \rightarrow \Re$ *is the function that gives the payoff for agent i for the final state of the game if each agent follows its corresponding strategy in the strategy profile s.*

A nash equilibrium for a game is a strategy profile decided upon by each player before the beginning of the game, such that no player can switch strategy and gain a guaranteed improvement in the payoff of the final history (for all possible choices made by the other players). I.e. it is a guaranteed maximum minimum payoff. Formally:

Definition 7.4. *Suppose* $T_i = T_i(s)$ *are the utility functions for a game* $G = \langle N, H, P, (\geq_i)_{i \in N} \rangle$*, a strategy profile S is said to be a* ***Nash equilibrium*** *of the game iff:*

$$\forall i \in N, \forall S_i \text{ alternatives strategies for player i, } T_i(S) \geq T_i(S^*)$$

where S^* *is the strategy profile obtained by replacing i's strategy in S by an alternative strategy* S_i*.*

Even though, as a general rule Nash equilibria have a natural correspondence to the best strategy to adopt, it is sometimes the case that it shows undesirable properties, since it does not take into account the sequential structure of the game; consider the following example.

EXAMPLE 7.1. Duel. *Sir Challenger was singing in praise to his beloved, lady Nimue. Sir Offender denied Nimue's absolute beauty and Challenger considers demanding satisfaction. If he does, Sir Offender will have to either accept or reject the demand. For Sir Challenger, honour is more important than life, while Sir Offender is indifferent, death and dishonour are the same thing. It is up to Sir Challenger to choose weapon: each gentleman feels confident that he will win if they use swords and both think they will perish if pistols are used. Finally, Sir Challenger is very brave but he would prefer for Offender to reject the duel than to die himself!* . This situation can be viewed as a strategic game (see Figure 7.1) with Nash equilibria: $\{\{challenge, pistols\}, \{reject\}\}$ and $\{\{challenge, swords\}, \{accept\}\}$.

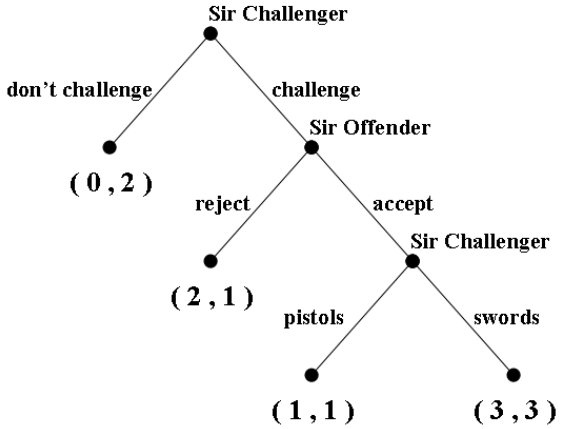


Figure 7.1: Strategic game: Duel.

Evidently, this yields a non desirable result ($\{\{challenge, pistols\}, \{reject\}\}$) as Sir Challenger would never be tempted to choose pistols. Therefore we seek a different approach by which the sequential structure of the game is decomposed, i.e.: players must be able to revise their strategies at every step in the game. This new perspective is captured by the notion of a subgame and a subgame perfect equilibrium.

Definition 7.5. *Given a game* $G = \langle N, H, P, (\geq_i)_{i \in N} \rangle$ *and a non-terminal history* h*, a* ***subgame*** *with respect to* h *is the game*

$$G|_h = \langle N, H|_h, P, (\geq_i)_{i \in N} \rangle$$

where $H|_h = \{k \in H \mid h \subseteq k\}$.

Given this definition, we will now seek strategy profiles which satisfy that they are a Nash equilibrium for every subgame of a given game (an optimal strategy in every possible situation). Such strategy profiles will be called **subgame perfect equilibria**. Subgames for example 7.1 are shown in Figure 7.2, this example has a unique subgame perfect equilibrium: $\{\{challenge, swords\}, \{accept\}\}$.

7.3 Extensive games as OCLPAS

While it is worth mentioning that [De Vos & Vermeir 2001] describe a way of representing a strategic game as a single OCLP we are interested in the final result of [De Vos & Vermeir 2001], according to which each player is an autonomous agent in a multi-agent system. There are two possible transformations: one to obtain the Nash equilibria and one to obtain the subgame perfect equilibria. The first transformation,

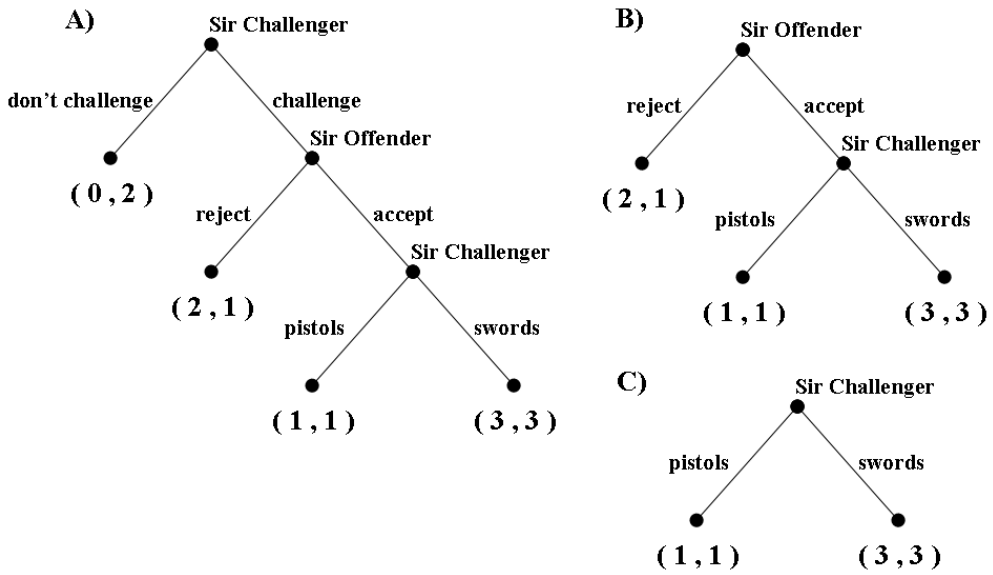


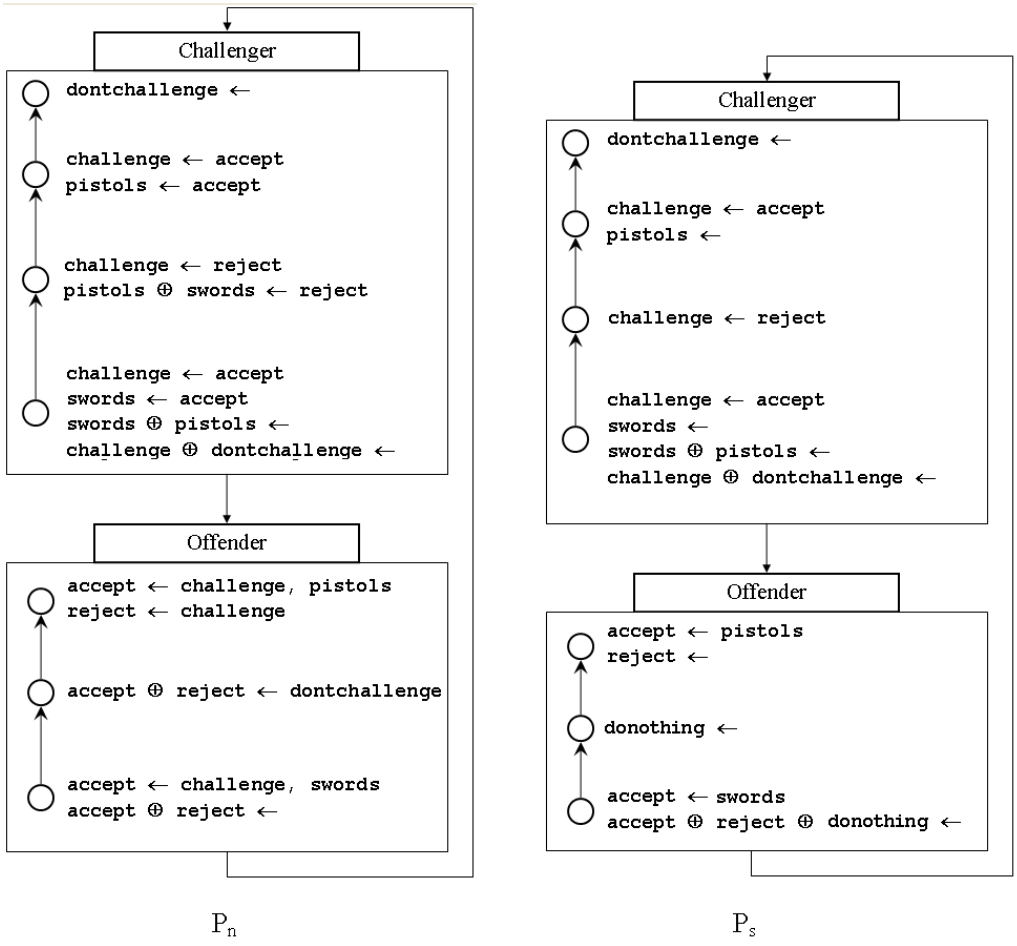
Figure 7.2: Subgames for the game: Duel.

denoted P_n needs to create an OCLPAS where:

- each agent represents a player.
- agent A's unique valid output is the agent whose turn is right after A's turn.
- for every possible payoff for a player A, there will be a component in its corresponding agent's internal OCLP with the following properties:
  - for every terminal history that has this payoff, the component will have a set of rules corresponding to that history. Each one of these sets will have a rule per decision made by the player, the rule's head corresponds to the agent's decision where past and future decisions by other players for this history are included in the body of the rule.
  - the component shall be more preferred than components corresponding to lesser payoff.

The second transformation, denoted P_s is very similar to the previous one, except that, since agents will only focus in the subgames they need only consider the future in the bodies of the rules. The OCLPAS for example 7.1 corresponding to both transformations is shown in Figure 7.3.

The fact that each agent is submitting one of her answer sets to the next player in the round can be interpreted as them thinking "What would the next player do if I do this? And what would the next one do in turn?".



P_n P_s

Figure 7.3: OCLPAS transformations for the game: Duel.

7.4 Demonstration

Using example 7.1 as motivation, the use of TOADs is demonstrated. Suppose that files challenger.oclp and offender.oclp (corresponding to the P_n transformation) are in the current folder (see Figure 7.4). Then in separate shells we run two agents, one for each program, making each of them both the valid input and valid output of the other. We specify that Challenger will be the starting agent and make use the queen to run the system in retained mode and observe it evolution. We proceed similarly for files SUBchallenger.oclp and SUBoffender.oclp (corresponding to the P_s transformation).

The output produced is shown in Appendix A.

7.5 Observation

We observe that contrary to our prediction, Nash equilibria are not calculated as fix-points although they are obtained after a single cycle. This doesn't contradict [De Vos 2001] theorems; indeed the stable models for this system are the nash equilibria, however since every agent is considering only one of the incoming answer sets, the system can loop and retrieve only one of the answers. The fix-point of the systems is a general agreement among all agents, but not all possible agreements.

After reiterated testing, we also observe that in spite of the non-determinism in the selection of answer sets, the evolutionary fix-point (after 5 cycles) of the OCLPAS is achieved always with $\{\{challenge, swords\}, \{accept\}\}$ (the subgame perfect equilibrium) as output.

Chapter 8

Conclusion

In this dissertation, the use of OCLP in agents was demonstrated and a tool called TOADs was developed in order to have a framework for the creation of reusable multi-agent systems with OCLP minded agents. TOADs succeeds in exhibiting the behaviour of this kind of agents and creates a needed bridge between Ordered Choice Logic Programming and Agent Oriented Programming (AOP); this bridge consists of: an ontology for OCLP and OCLPAS in JADE, a collection of JADE behaviour classes for each of the roles defined in TOADs and a set of static methods included in the OCLPAgentTools class.

Towards the end of this project the use of TOADs was illustrated and general guidance was given for both programmers and logicians to experiment, highlighting the most relevant features and services.

Finally a demonstration of the system was given exemplifying its application to Game Theory, materialising the work of [De Vos 2001].

As a conclusion, the developer wishes to commentate on the state of the application, give a critical review of its implementation including known flaws as well as suggesting possible enhancements to be made.

8.1 Not yet implemented features

Due to time constraints, no logger tool was implemented. A standard output is produced however and the user may choose to redirect this to a file.

Furthermore, [De Vos 2001] highlights the need to treat carefully OCLPAS with internal cycles, TOADs does not contemplate this situation and an implementation of sub-framework isolation is left as a future enhancement.

8.2 Critique

The final product satisfies most of the specification and it has been tested to work according to what is expected (with the exception of cyclic subsystems); however

many implementation solutions are far from optimal.

The program fails to implement an absolutely independent set of behaviours. Several among them interact actively on each other, accessing their ample public interfaces, which should be restricted for security.

Graphical interfaces are lacking in style and usability since they were not intended to abide by any HCI standard whatsoever; nonetheless these were built in a 2-layer fashion so that in the future a better graphical layer could be used.

Finally, all GUIs were implemented directly by invoking swing components from a new thread, different from the agent thread. According to JADE documentation, this is not advisable and it is recommended that agents should extend the GuiAgent class provided by JADE which gives means of a proper interaction between threads so that methods in a thread are not actively invoked from another thread.

8.2.1 Known errors and issues

Even though efforts have been made to ensure robustness by implementing the CFAS protocol, there will still be cases in which the system could show an unwanted behaviour. Specifically, suppose agent A issues a CFAS to agent B; agent B is not responding -for unknown reasons- and agent A's deadline expires. Agent A will update with other agents' incoming data, however agent A herself will not respond to her CFAS for all the time she was waiting for agent B's response. Evidently this may cause a cascade effect inhibiting an entire sub-framework of the system. Therefore it is highly recommended that deadlines for each agent are adjusted carefully, considering its life cycle duration and its inputs deadlines. As a general rule we recommend that if agent A is a valid input for agent B, then the deadline for replies to CFAS issued by agent B should be at least X seconds longer than the one for A, where X is the estimated time for an ACLMessage to reach B from A (dependent on the network).

8.3 Further research and enhancements suggestions

Error and timeout handling were not implemented and remain as future features. An future version could include syntax expansions to include these and function calls.

Furthermore it would be interesting to analyse if it is efficient to compute not one, but all of the incoming answer sets in a combinatorial fashion, splitting an agent into several threads. We conjecture that such a procedure would improve the computational time needed to find an evolutionary fix-point for a given OCLPAS.

Due to the nature of this framework, it would be interesting to see how well it adapts to the needs of artificial life programming, as the domain requires preference and choice to be represented clearly.

Chapter 9

Bibliography

[**Aart et al 2002.**] van Aart, C.J., Pels, R.F., Giovanni C. and Bergenti F. Creating and Using Ontologies in Agent Communication. Workshop on Ontologies in Agent Systems 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, 2002.
[**Alferes et al 1998.**] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, T. Przymusinski. Dynamic logic programming. In: A. Cohn and L. Schubert, editors, KR '98, pp. 98-109. Morgan Kaufmann, 1998.
[**Baumann et al 2003.**] J. Baumann, F. Hohl, K. Rothermel, M. Straer. Mole - Concepts of a Mobile Agent System. [Online] Available at: http://citeseer.nj.nec.com/baumann97mole.html (accessed December 2003).
[**Belavkin 2003.**] Belavkin, R. V. (2003). On emotion, learning and uncertainty: a cognitive modelling approach. PhD Thesis, The University of Nottingham, UK.
[**Bellifemine, Poggi & Rimassa 2001.**] Bellifemine, F., Poggi, A., Rimassa, G. Developing multi-agent systems with a FIPA-compliant agent framework. In: Software - Practice And Experience, 2001 no. 31, pp. 103-128.
[**Brain & De Vos 2003.**] Brain M, de Vos, M. Implementing OCLP as a front-end for Answer Set Solvers: From Theory to Practice. In ASP03: Answer Set Programming: Advances in Theory and Implementation. Ceur-WS, September 2003. [Online] Available at http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-78/asp03-final-brain.pdf
[**Bratman 1987.**] M. E. Bratman. Intention, Plans, and Practical Reason. Harvard University Press: Cambridge, MA, 1987.
[**Bratman 1990.**] M. E. Bratman. What is intention? In P. R. Cohen, J. L. Morgan, and M. E. Pollack, editors, Intentions in Communication, pages 15[32. The MIT Press: Cambridge, MA, 1990.
[**Christof 1999.**] Christof Monz (1999). Modeling Ambiguity in a Multi-Agent System. In: P. Dekker (Ed.) Proceedings of the 12th Amsterdam Colloquium (AC'99). Institute for Logic, Language and Computation, 1999, pages 43-48.
[**Cohen & Levesque 1990.**] P. R. Cohen and H. J. Levesque. Intention is choice

with commitment. Artificial Intelligence, 42:213-261, 1990.
[**Damasio 1994.**] Antonio R. Damasio, Descartes' Error: Emotion, Reason, and the Human Brain, New York: G.P. Putnam, 1994.
[**De Vos 2001.**] Marina de Vos. Logic Programming, Decisions and Games. PhD Thesis. Vrije Universitiet Brussel.
[**De Vos & Vermeir 1999.**] Marina De Vos and Dirk Vermeir. Choice Logic Programs and Nash Equilibria in Strategic Games. In Jorg Flum and Mario Rodrguez-Artalejo, editors, Computer Science Logic (CSL'99), volume 1683 of Lecture Notes in Computer Science, pages 266-276, Madrid, Spain, 1999. Springer Verslag.
[**De Vos & Vermeir 2001.**] Marina de Vos and Dirk Vermeir. Logic Programming Agents and Game Theory. In Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, pp. 27–33. American Association for Artificial Intelligence Press, Stanford (Palo Alto), California, US, 2001.
[**De Vos & Vermeir 2002.**] Marina de Vos and Dirk Vermeir. Dynamic Decision Making in Logic Programming and Game Theory. In AI2002: Advances in Artificial Intelligence, Lecture Notes in Artificial Intelligence, pages 36-47. Springer, December 2002.
[**De Vos & Vermeir 2003.**] Marina de Vos and Dirk Vermeir. Technical Report: Using Implicit Negation for the Computation of Skeptical and Credulous Answer Sets of Ordered Choice Logic Programs. Submitted for the special issue on preferences in AI and CP of the Journal of Computational Intelligence, 2003.
[**Dell'Acqua, Leite & Pereira 2001.**] P. Dell'Acqua, J. A. Leite and L. M. Pereira, Evolving Multi-Agent Viewpoints - an architecture. In P. Brazdil and A. Jorge (eds.), Progress in Artificial Intelligence, 10th Portuguese International Conference on Artificial Intelligence (EPIA'01), Springer, LNAI 2258, Porto, December 2001.
[**Dubois, Lang & Prade 1994.**] D. Dubois, J. Lang, and H. Prade. Possibilistic Logic, pages 439-503. Clarendon Press, Oxford, 1994.
[**El-Nasr & Yen 1998.**] Magy Seif El-Nasr and John Yen, "Agents, Emotional Intelligence and Fuzzy Logic" Proc. of the 17th Annual Meeting of the North American Fuzzy Information , 1998.
[**Geltkin & Arslan 2002.**] Gltekin, I, Arslan, A. (2002). Modular-Fuzzy Cooperation Algorithm For Multi-Agent Systems. In: Lecture Notes in Computer Science, vol. 2457, pp. 255 - 263.
[**Georgeff & Lansky 1987.**] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), pages 677[682, Seatt le, WA, 1987.
[**Hoek & Wooldridge 2003.**] W. van der Hoek and W. Wooldrige (2003). Towards a Logic of Rational Agency. Logic Journal of the IGPL 11(2), pp. 133-157.
[**Huth & Ryan 2000.**] Huth, M. Ryan, M (2000, reprinted 2001, 2002). Logic in Computer Science: Modelling and reasoning about systems. Cambridge: Cambridge University Press.
[**Osborne & Runinstein.**] Osborne, M. J., Rubinstein, A. 1996. A Course in Game

Theory. Cambridge, Massachusets, London, England: The MIT Press, third edition.
[**Leite et al 2000.**] J. A. Leite, J. J. Alferes and L. M. Pereira, Multi-dimensional Dynamic Logic Programming, In F. Sadri and K. Satoh (eds.), Procs. of the CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA'00), pages 17-26, London, England, July 2000.
[**Malpas 1987.**] Malpas J. PROLOG ? A Relational Language and its Applications. Prentice-Hall International, Englewood Cliffs, New Jersey, 1987.
[**Milojicic et al 1998.**] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF: The OMG mobile agent system interoperability facility. In Proceedings of Mobile Agents, pages 50-67, Stuttgart, Germany, Sept. 1998.
[**Patil et al 1992.**] Patil R, Fikes R., Patel-Schneider P., McKay, D, Finin T, Gruber T., Neches, R. The DARPA Knowledge Sharing Effort: Progress Report. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference, Cambridge, MA, 1992; 103-114.
[**Prot́[e]ǵ[e].**] Prot́[e]ǵ[e] user guide. [Online] Available at: http://protege.stanford.edu/publications/UserGuide.pdf
[**Rao & Georgeff 1995.**] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. Proceedings of the First Intl. Conference on Multiagent Systems, San Francisco, CA, 1995; pp. 312-319.
[**Richardson 2003.**] Richardson, D. Formal systems, logic and semantics. Lecture notes on Computer Science. [Online] Available at http://www.bath.ac.uk/ masdr/c19.ps
[**Searl 1969.**] Searle, J. R. (1969), Speech Acts. An Essay in the Philosophy of Language. Cambridge.
[**Somerville 2001.**] Sommerville, I. Software Engineering. 6th edition. Addison-Wesley Publishers Limited 1989, 2001.
[**Sycara et al 1996.**] K. Sycara, K. Decker, A. Pannu, M. Williamson and D. Zeng. Distributed intelligent agents. IEEE Expert 1996; 11(6); 36-46.
[**Turing 1936.**] Turing, A.M. 1936. 'On Computable Numbers, with an Application to the Entscheidungsproblem'. Proceedings of the London Mathematical Society, Series 2, 42 (1936-37), pp.230-265.
[**Vidal 2003.**] Vidal, J. M.. (2003). FIPA Introduction. [Online] Available at: http://jmvidal.cse.sc.edu/talks/fipaintro/allslides.pdf (accessed December 4 2003).
[**Wooldridge 2002.**] M. Wooldridge. An Introduction to Multiagent Systems. John Wiley and Sons Ltd, February 2002.
[**Zadeh 1965.**] Zadeh L.A. (1965) Fuzzy sets. Information and Control, 8, 338-353.

A.1 Duel game: Challenger's output

```
[tadeo@rpc-ma1tmc 0.1.0]$ /usr/java/j2sdk1.4.2_03/bin/java
jade.Boot -port 1234 -container
"Challenger:worldpeace.toads.OCLPAgent(queen:Queen@RP
C-
MA1TMC:1234/JADE,open:challenger.oclp,input:Offender,o
utput:Offender,semantics:c,starter:true)"
   This is JADE 3.1 - 2003/12/17 13:40:15
   downloaded in Open Source, under LGPL restrictions,
   at http://jade.cselt.it/

Toads jumping all around! Toad running version Beta
1005.0025
...parsing parameters: queen:Queen@RPC-
MA1TMC:1234/JADE,open:challenger.oclp,input:Offender,o
utput:Offender,semantics:c,starter:true
>> Accepting Queen@RPC-MA1TMC:1234/JADE as my
Queen!
Read file challenger.oclp. Program is
component dishonour {
dontchallenge.
}

component death {
challenge :- accept.
pistols :- accept.
}

component rejected {
challenge :- reject.
pistols + swords :- reject.
}

component swordfight {
challenge :- accept.
swords :- accept.
swords + pistols.
challenge + dontchallenge.
}

swordfight < rejected
rejected < death
death < dishonour

Agent container Container-31@JADE-IMTP://rpc-ma1tmc is
ready.
File read and interpreted as:
component dishonour {
dontchallenge.
}

component death {
challenge :- accept.
pistols :- accept.
}

component rejected {
challenge :- reject.
pistols + swords :- reject.
}

component swordfight {
challenge :- accept.
swords :- accept.
swords + pistols.
challenge + dontchallenge.
}


swordfight < rejected
rejected < death
death < dishonour

Adding the local agent Offender to my valid inputs...
Adding the local agent Offender to my valid outputs...
Running in Retained Mode.
OCT output is: Stable Model: dontchallenge pistols  -------
parsed and recognised as{ dontchallenge, pistols }
```

OCT output is: Stable Model: dontchallenge swords ------- parsed and recognised as{ dontchallenge, swords }
Output prepared:
{ dontchallenge, pistols }
{ dontchallenge, swords }

Challenger: started.Running in Retained Mode.
Received a CFAS message from Offender@rpc-ma1tmc:1234/JADE Cycle:1. Reply is
{ dontchallenge, pistols }
{ dontchallenge, swords }

Challenger: preparing to broadcast message: { dontchallenge, pistols }
{ dontchallenge, swords }

Received a proper OCLP_message message, content: { pistols, reject, dontchallenge }
{ accept, pistols, dontchallenge }
All messages received. Proceeding to filter the input
OCT output is: A0A0A0A0A0Stable Model: challenge pistols reject ------- parsed and recognised as{ challenge, pistols, reject }
Solutions for this cycle are:
{ challenge, pistols, reject }

CFAS timer started: agent shall wait for CFAS's for the next 7 seconds.
CFAS timer elapsed.
Received a CFAS message from Offender@rpc-ma1tmc:1234/JADE Cycle:2. Reply is
{ challenge, pistols, reject }

Challenger: preparing to broadcast message: { challenge, pistols, reject }

CFAS timer was terminated before timeout expired.
Received a proper OCLP_message message, content: { accept, challenge, pistols }
{ challenge, pistols, reject }
All messages received. Proceeding to filter the input
OCT output is: A1A1A1A1A1Stable Model: challenge accept swords ------- parsed and recognised as{ challenge, accept, swords }
Solutions for this cycle are:
{ challenge, accept, swords }

CFAS timer started: agent shall wait for CFAS's for the next 7 seconds.
CFAS timer elapsed.
Received a CFAS message from Offender@rpc-ma1tmc:1234/JADE Cycle:3. Reply is
{ challenge, accept, swords }

Challenger: preparing to broadcast message: { challenge, accept, swords }

CFAS timer was terminated before timeout expired.
Received a proper OCLP_message message, content: { accept, challenge, swords }
All messages received. Proceeding to filter the input
OCT output is: A2A2A2A2A2Stable Model: challenge accept swords ------- parsed and recognised as{ challenge, accept, swords }
Solutions for this cycle are:
{ challenge, accept, swords }

CFAS timer started: agent shall wait for CFAS's for the next 7 seconds.
CFAS timer elapsed.
Received a CFAS message from Offender@rpc-ma1tmc:1234/JADE Cycle:4. Reply is
{ challenge, accept, swords }

Challenger: preparing to broadcast message: { challenge, accept, swords }

CFAS timer was terminated before timeout expired.

Received a proper OCLP_message message, content: { accept, challenge, swords }
All messages received. Proceeding to filter the input
OCT output is: A3A3A3A3A3Stable Model: challenge accept swords ------- parsed and recognised as{ challenge, accept, swords }
Solutions for this cycle are:
{ challenge, accept, swords }

Challenger: achieved evolutionary fix-point. Switching to retained mode.
CFAS timer started: agent shall wait for CFAS's for the next 7 seconds.
CFAS timer elapsed.

A.2 Duel game: Offender's output

```
[tadeo@rpc-ma1tmc 0.1.0]$ /usr/java/j2sdk1.4.2_03/bin/java
jade.Boot -port 1234 –container
"Offender:worldpeace.toads.OCLPAgent(queen:Queen@RPC
-
MA1TMC:1234/JADE,open:offender.oclp,input:Challenger,o
utput:Challenger,semantics:c)"
    This is JADE 3.1 - 2003/12/17 13:40:15
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.cselt.it/

Toads jumping all around! Toad running version Beta
1005.0025
...parsing parameters: queen:Queen@RPC-
MA1TMC:1234/JADE,open:offender.oclp,input:Challenger,o
utput:Challenger,semantics:c
Agent container Container-32@JADE-IMTP://rpc-ma1tmc is
ready.
>> Accepting Queen@RPC-MA1TMC:1234/JADE as my
Queen!
Read file offender.oclp. Program is
component deathorshame {
accept :- challenge, pistols.
reject :- challenge.
}

component noduel {
accept + reject :- dontchallenge.
}

component swordfight {
accept :- challenge, swords.
accept + reject.
}

swordfight < noduel
noduel < deathorshame

File read and interpreted as:
component deathorshame {
accept :- challenge, pistols.
reject :- challenge.
}

component noduel {
accept + reject :- dontchallenge.
}

component swordfight {
accept :- challenge, swords.
accept + reject.
}


swordfight < noduel
noduel < deathorshame

Adding the local agent Challenger to my valid inputs...
Adding the local agent Challenger to my valid outputs...
Running in Retained Mode.
OCT output is: Stable Model: reject  ------- parsed and
recognised as{ reject }
OCT output is: Stable Model: accept  ------- parsed and
recognised as{ accept }
Output prepared:
{ reject }
{ accept }

Offender: started.Running in Retained Mode.
Received a CFAS message from Challenger@rpc-
ma1tmc:1234/JADE Cycle:1. Storing CFAS.
Reason: CFAS cycle was: 1. CurrentCycle is: 0
prepareResultNotification() method not re-defined
Received a proper OCLP_message message, content: {
dontchallenge, pistols }
{ dontchallenge, swords }
All messages received. Proceeding to filter the input
OCT output is: A0A0A0A0Stable Model: pistols reject
dontchallenge  ------- parsed and recognised as{ pistols, reject,
dontchallenge }
```

OCT output is: Stable Model: accept pistols dontchallenge ---
---- parsed and recognised as{ accept, pistols, dontchallenge }
Solutions for this cycle are:
{ pistols, reject, dontchallenge }
{ accept, pistols, dontchallenge }

For cycle: 1Sending reply (OCLP_message) to pending
CFAS, and removing it from list.
Offender: preparing to broadcast message: { pistols, reject,
dontchallenge }
{ accept, pistols, dontchallenge }

CFAS timer started: agent shall wait for CFAS's for the next
7 seconds.
CFAS timer elapsed.
Received a CFAS message from Challenger@rpc-
ma1tmc:1234/JADE Cycle:2. Storing CFAS.
Reason: CFAS cycle was: 2. CurrentCycle is: 1
prepareResultNotification() method not re-defined
Received a proper OCLP_message message, content: {
challenge, pistols, reject }
All messages received. Proceeding to filter the input
OCT output is: A1A1A1A1Stable Model: accept challenge
pistols ------- parsed and recognised as{ accept, challenge,
pistols }
OCT output is: Stable Model: challenge pistols reject -------
parsed and recognised as{ challenge, pistols, reject }
Solutions for this cycle are:
{ accept, challenge, pistols }
{ challenge, pistols, reject }

For cycle: 2Sending reply (OCLP_message) to pending
CFAS, and removing it from list.
Offender: preparing to broadcast message: { accept, challenge,
pistols }
{ challenge, pistols, reject }

CFAS timer started: agent shall wait for CFAS's for the next
7 seconds.
CFAS timer elapsed.
Received a CFAS message from Challenger@rpc-
ma1tmc:1234/JADE Cycle:3. Storing CFAS.
Reason: CFAS cycle was: 3. CurrentCycle is: 2
prepareResultNotification() method not re-defined
Received a proper OCLP_message message, content: {
challenge, accept, swords }
All messages received. Proceeding to filter the input
OCT output is: A2A2A2A2Stable Model: accept challenge
swords ------- parsed and recognised as{ accept, challenge,
swords }
Solutions for this cycle are:
{ accept, challenge, swords }

For cycle: 3Sending reply (OCLP_message) to pending
CFAS, and removing it from list.
Offender: preparing to broadcast message: { accept, challenge,
swords }

CFAS timer started: agent shall wait for CFAS's for the next
7 seconds.
CFAS timer elapsed.
Received a CFAS message from Challenger@rpc-
ma1tmc:1234/JADE Cycle:4. Storing CFAS.
Reason: CFAS cycle was: 4. CurrentCycle is: 3
prepareResultNotification() method not re-defined
Received a proper OCLP_message message, content: {
challenge, accept, swords }
All messages received. Proceeding to filter the input
OCT output is: A3A3A3A3Stable Model: accept challenge
swords ------- parsed and recognised as{ accept, challenge,
swords }
Solutions for this cycle are:
{ accept, challenge, swords }

For cycle: 4Sending reply (OCLP_message) to pending
CFAS, and removing it from list.
Offender: preparing to broadcast message: { accept, challenge,
swords }

CFAS timer started: agent shall wait for CFAS's for the next
7 seconds.

A.3 Duel Subgame: Challenger's output

```
[tadeo@rpc-ma1tmc 0.1.0]$
/usr/java/j2sdk1.4.2_03/bin/java
jade.Boot -port 1234 -container
"SUBChallenger:worldpeace.toads.OCLPAgent
(queen:Queen@RPC-
MA1TMC:1234/JADE,open:SUBchallenger.oclp,
input:SUBOffender,output:SUBOffender,sema
ntics:c,starter:true)"
    This is JADE 3.1 - 2003/12/17
13:40:15
    downloaded in Open Source, under LGPL
restrictions,
    at http://jade.cselt.it/

Toads jumping all around! Toad running
version Beta 1005.0025
...parsing parameters: queen:Queen@RPC-
MA1TMC:1234/JADE,open:SUBchallenger.oclp,
input:SUBOffender,output:SUBOffender,sema
ntics:c,starter:true
>> Accepting Queen@RPC-MA1TMC:1234/JADE
as my Queen!
Read file SUBchallenger.oclp. Program is
component dishonour {
dontchallenge.
}

component death {
challenge :- accept.
pistols.
}

component rejected {
challenge :- reject.
}

component swordfight {
challenge :- accept.
swords.
swords + pistols.
challenge + dontchallenge.
}

swordfight < rejected
rejected < death
death < dishonour

File read and interpreted as:
component dishonour {
dontchallenge.
}

component death {
challenge :- accept.
pistols.
}

component rejected {
challenge :- reject.
}

component swordfight {
challenge :- accept.
swords.
swords + pistols.
challenge + dontchallenge.
}


swordfight < rejected
rejected < death
death < dishonour

Adding the local agent SUBOffender to my
valid inputs...
Adding the local agent SUBOffender to my
valid outputs...
Running in Retained Mode.
Agent container Container-27@JADE-
IMTP://rpc-ma1tmc is ready.
```

```
OCT output is: Stable Model:
dontchallenge swords  ------- parsed and
recognised as{ dontchallenge, swords }
Output prepared:
{ dontchallenge, swords }

SUBChallenger: started.Running in
Retained Mode.
Received a CFAS message from
SUBOffender@rpc-ma1tmc:1234/JADE Cycle:1.
Reply is
 { dontchallenge, swords }

SUBChallenger: preparing to broadcast
message: { dontchallenge, swords }

Received a proper OCLP_message message,
content: { accept, swords, dontchallenge
}
All messages received. Proceeding to
filter the input
OCT output is: A0A0A0A0A0Stable Model:
challenge accept swords  ------- parsed
and recognised as{ challenge, accept,
swords }
Solutions for this cycle are:
{ challenge, accept, swords }


# CFAS timer started: agent shall wait
for CFAS's for the next 7 seconds.
# CFAS timer elapsed.
Received a CFAS message from
SUBOffender@rpc-ma1tmc:1234/JADE Cycle:2.
Reply is
 { challenge, accept, swords }

SUBChallenger: preparing to broadcast
message: { challenge, accept, swords }

# CFAS timer was terminated before
timeout expired.
Received a proper OCLP_message message,
content: { accept, swords, challenge }
All messages received. Proceeding to
filter the input
OCT output is: A1A1A1A1A1Stable Model:
challenge accept swords  ------- parsed
and recognised as{ challenge, accept,
swords }
Solutions for this cycle are:
{ challenge, accept, swords }


# CFAS timer started: agent shall wait
for CFAS's for the next 7 seconds.
# CFAS timer elapsed.
Received a CFAS message from
SUBOffender@rpc-ma1tmc:1234/JADE Cycle:3.
Reply is
 { challenge, accept, swords }

SUBChallenger: preparing to broadcast
message: { challenge, accept, swords }

# CFAS timer was terminated before
timeout expired.
Received a proper OCLP_message message,
content: { accept, swords, challenge }
All messages received. Proceeding to
filter the input
OCT output is: A2A2A2A2A2Stable Model:
challenge accept swords  ------- parsed
and recognised as{ challenge, accept,
swords }
Solutions for this cycle are:
{ challenge, accept, swords }


SUBChallenger: achieved evolutionary fix-
point. Switching to retained mode.
# CFAS timer started: agent shall wait
for CFAS's for the next 7 seconds.
# CFAS timer elapsed.
```

A.4 Duel Subgame: Offender's output

```
[tadeo@rpc-ma1tmc 0.1.0]$
/usr/java/j2sdk1.4.2_03/bin/java
jade.Boot -port 1234 -container
"SUBOffender:worldpeace.toads.OCLPAgent(q
ueen:Queen@RPC-
MA1TMC:1234/JADE,open:SUBoffender.oclp,in
put:SUBChallenger,output:SUBChallenger,se
mantics:c)"
    This is JADE 3.1 - 2003/12/17
13:40:15
    downloaded in Open Source, under LGPL
restrictions,
    at http://jade.cselt.it/

Toads jumping all around! Toad running
version Beta 1005.0025
...parsing parameters: queen:Queen@RPC-
MA1TMC:1234/JADE,open:SUBoffender.oclp,in
put:SUBChallenger,output:SUBChallenger,se
mantics:c
>> Accepting Queen@RPC-MA1TMC:1234/JADE
as my Queen!
Read file SUBoffender.oclp. Program is
component deathorshame {
accept :- pistols.
reject.
}

component noduel {
donothing.
}

component swordfight {
accept :- swords.
accept + reject + donothing.
}

swordfight < noduel
noduel < deathorshame

Agent container Container-26@JADE-
IMTP://rpc-ma1tmc is ready.
File read and interpreted as:
component deathorshame {
accept :- pistols.
reject.
}

component noduel {
donothing.
}

component swordfight {
accept :- swords.
accept + reject + donothing.
}


swordfight < noduel
noduel < deathorshame

Adding the local agent SUBChallenger to
my valid inputs...
Adding the local agent SUBChallenger to
my valid outputs...
Running in Retained Mode.
OCT output is: Stable Model: donothing  -
------ parsed and recognised as{
donothing }
Output prepared:
{ donothing }

SUBOffender: started.Running in Retained
Mode.
Received a CFAS message from
SUBChallenger@rpc-ma1tmc:1234/JADE
Cycle:1. Storing CFAS.
Reason: CFAS cycle was: 1. CurrentCycle
is: 0
```

```
prepareResultNotification() method not
re-defined
Received a proper OCLP_message message,
content: { dontchallenge, swords }
All messages received. Proceeding to
filter the input
OCT output is: A0A0A0A0Stable Model:
accept swords dontchallenge  -------
parsed and recognised as{ accept, swords,
dontchallenge }
Solutions for this cycle are:
{ accept, swords, dontchallenge }


For cycle: 1Sending reply (OCLP_message)
to pending CFAS, and removing it from
list.
SUBOffender: preparing to broadcast
message: { accept, swords, dontchallenge
}

# CFAS timer started: agent shall wait
for CFAS's for the next 7 seconds.
# CFAS timer elapsed.
Received a CFAS message from
SUBChallenger@rpc-ma1tmc:1234/JADE
Cycle:2. Storing CFAS.
Reason: CFAS cycle was: 2. CurrentCycle
is: 1
prepareResultNotification() method not
re-defined
Received a proper OCLP_message message,
content: { challenge, accept, swords }
All messages received. Proceeding to
filter the input
OCT output is: A1A1A1A1Stable Model:
accept swords challenge  ------- parsed
and recognised as{ accept, swords,
challenge }
Solutions for this cycle are:
{ accept, swords, challenge }


For cycle: 2Sending reply (OCLP_message)
to pending CFAS, and removing it from
list.
SUBOffender: preparing to broadcast
message: { accept, swords, challenge }

# CFAS timer started: agent shall wait
for CFAS's for the next 7 seconds.
# CFAS timer elapsed.
Received a CFAS message from
SUBChallenger@rpc-ma1tmc:1234/JADE
Cycle:3. Storing CFAS.
Reason: CFAS cycle was: 3. CurrentCycle
is: 2
prepareResultNotification() method not
re-defined
Received a proper OCLP_message message,
content: { challenge, accept, swords }
All messages received. Proceeding to
filter the input
OCT output is: A2A2A2A2Stable Model:
accept swords challenge  ------- parsed
and recognised as{ accept, swords,
challenge }
Solutions for this cycle are:
{ accept, swords, challenge }


For cycle: 3Sending reply (OCLP_message)
to pending CFAS, and removing it from
list.
SUBOffender: preparing to broadcast
message: { accept, swords, challenge }

# CFAS timer started: agent shall wait
for CFAS's for the next 7 seconds.
# CFAS timer elapsed.
```

B.1 OCLPAgent.java

```
package worldpeace.toads;

import jade.core.*;
import jade.core.behaviours.*;
import jade.proto.AchieveREResponder;
import jade.proto.AchieveREInitiator;
import jade.proto.SimpleAchieveREResponder;
import jade.lang.acl.*;
import toads.ontology.*;
// import java.util.*;
import java.io.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.Arrays;
import java.util.Date;

import jade.util.leap.*;

import jade.content.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import jade.content.lang.*;
import jade.content.lang.sl.*;

import jade.domain.FIPANames.InteractionProtocol;

import worldpeace.toads.OCLPAgentTools;
import worldpeace.toads.CFASTimerBehaviour;
import worldpeace.toads.FacilitateInfoBehaviour;
import worldpeace.toads.UpdateBehaviour;
import worldpeace.toads.SubscriptionHandlerBehaviour;
import worldpeace.toads.RetainedBehaviour;
import worldpeace.toads.MainEngineBehaviour;

public class OCLPAgent extends Agent {

    private AID theQueen = new AID( "Queen", AID.ISLOCALNAME );
    //Hardcoded default Queen name, as "Queen" in local platform.

    private AnswerSet input = new AnswerSet();
    private AnswerSets lastOutput = null;
    // Used to rememeber last message sent.

    private boolean evolutionaryFlag = true;
    // Used to identify the evolutionary fix-point. Will be true if the previous cycle's .
    // Answer Sets are equivalent to the current cycle's ones.

    private boolean lastEvolutionaryFlag = false;
    // Used to identify the evolutionary fix-point. Holds the value of the last cycle's evolutionary flag.
    // If this flag is true two cycles in a row, the system is in an evolutionary fix-point.

    private int currentcycle;
    // the current cycle

    private jade.util.leap.List lastOutputMessages = new ArrayList();
    // here we store messages to be broadcast when waiting, in retained Moded

    private boolean retainedMode = false;
    // true: retained mode active   false: runtime mode active

    private boolean isStartingAgent = false;
    // Determines whether this agent is a starting agent.
    // Starting agents are needed in cyclic systems to take initiative and start the cascade effect.
    // Starting agents will reply to CFAS for cycle 1 even when running cycle 0 so as to cast the
    // spark that lights the chain reaction

    private jade.util.leap.List CFASs = new ArrayList();
    // A record of the CFAS's received

    private long OCLP_message_TIMEOUT = 7000;
    // The agent will wait this long for CFAS to be answered

    private long CFAS_TIMEOUT = 7000;
    // The agent will wait this long for CFAS to arrive once the output is ready

    private CFASTimerBehaviour CFAStimer = null;

    public long CFASListeningTimer = 0;
    // Time at which last output was calculated

    private int numberCFASreplied = 0;
    // Number of OCLP_message messages sent. Needed to know when a cycle is finished

    private boolean cycleCompleted = true;
    // Flag stating the current cycle is completed and a new one can start when the Queen commands so

    private CommunicationChannels commChannels = new CommunicationChannels();
    // defines the communication channels, containing:
    // List channelInput = new ArrayList();
    //      A list of AID objects identifying the allowed input agents
    //
    // List channelOutput = new ArrayList();
    //      A list of AID objects identifying the output agents

    private OCLP_program brain = new OCLP_program();
    // the internal program

    private boolean credulous = false;
    // Semantics: false -> Skeptic    true -> Credulous

    private String OCT_PATH = "./oct";
    // Location of the answer set solver OCT

    private boolean subscription = false;
    // True if the queen is subscribed

    Codec language;
    Ontology ontology;

    public OCLPAgent(){ super(); }

    // Enhanced constructor
    public OCLPAgent(List input, List output, boolean retained){
                super();
                retainedMode = retained;
                List ci = new ArrayList();
                List co = new ArrayList();
                for (int i = 0; i<input.size(); i++)
                    ci.toArray()[i]=input.toArray()[i];
                for (int i = 0; i<output.size(); i++)
                    co.toArray()[i]=output.toArray()[i];
                commChannels.setChannelInput(ci);
                commChannels.setChannelOutput(co);
    }

    // Runtime mode interface
    public AID getTheQueen() { return theQueen; }
    public void setTheQueen(AID queenAgentAID) { theQueen = queenAgentAID; }
    public boolean getRetainedMode() { return retainedMode; }
    public void setRetainedMode(boolean mode) { retainedMode = mode; }
    public AnswerSet getInput() { return input; }
    public void setInput(AnswerSet inp) { input = inp; }
```

```
public boolean getEvolutionaryFlag() { return evolutionaryFlag; }
public void setEvolutionaryFlag(boolean flag) { evolutionaryFlag = flag; }
public boolean getLastEvolutionaryFlag() { return lastEvolutionaryFlag; }
public void setLastEvolutionaryFlag(boolean flag) { lastEvolutionaryFlag = flag; }
public int getCurrentcycle() { return currentcycle; }
public void setCurrentcycle(int cycle) { currentcycle = cycle; }
public jade.util.leap.List getLastOutputMessages() { return lastOutputMessages; }
public void setLastOutputMessages (jade.util.leap.List lom) { lastOutputMessages = lom; }
public OCLP_program getOCLP() { return brain; }
public void setOCLP( OCLP_program updatedOCLP ) { brain = updatedOCLP; }
public CommunicationChannels getCommChannels() { return commChannels; }
public void setCommChannels( CommunicationChannels channels ) { commChannels = channels; }
public boolean getIsStartingAgent() { return isStartingAgent; }
public void setIsStartingAgent(boolean isit) { isStartingAgent = isit; }
public boolean getSemantics() { return credulous; }
public boolean getCredulous() { return credulous; }
public boolean getSceptic() {return (!credulous); }
public void setSemantics(boolean c) { credulous = c; }
public void setCredulous() { credulous = true; }
public void setCredulous(boolean c) { credulous = c; }
public void setSceptic() { credulous = false; }
public void setSceptic(boolean c) { credulous = (!c); }
public AnswerSets getLastOutput() { return lastOutput; }
public void setLastOutput(AnswerSets loutput) { lastOutput = loutput; }
public String getOCT_PATH() { return OCT_PATH; }
public void setOCT_PATH(String octpath) { OCT_PATH = octpath; }
public jade.util.leap.List getCFASs() { return CFASs; }
public void setCFASs(jade.util.leap.List cfass) { CFASs = cfass; }
public boolean getCycleCompleted() { return cycleCompleted; }
public void setCycleCompleted(boolean cc) { cycleCompleted = cc; }
public int getNumberCFASreplied() { return numberCFASreplied; }
public void setNumberCFASreplied(int n) { numberCFASreplied = n; }
public long getOCLP_message_TIMEOUT() { return OCLP_message_TIMEOUT; }
public void setOCLP_message_TIMEOUT(long timeout) { OCLP_message_TIMEOUT = timeout; }
public long getCFAS_TIMEOUT() { return CFAS_TIMEOUT; }
public void setCFAS_TIMEOUT(long timeout) { CFAS_TIMEOUT = timeout; }
public void setSubscription(boolean subs) { subscription = subs; }
public boolean getSubscription() { return subscription; }
public Codec getLanguage() { return language; }
public void setLanguage(Codec l) { language = l; }
public Ontology getOntology() { return ontology; }
public void setOntology(Ontology o) { ontology = o; }

// Runtime mode event methods. These should be overrided to handle appropriate events:

public void TOADs_init() { return; }
// called when the agent is started. Programs should return the control
// to TOADs whenever they wish the system to start functioning

public boolean TOADs_achievedEvolutionaryFixPoint(int currentcycle, AnswerSets lastOutput) {
        System.out.println(this.getLocalName()+": achieved evolutionary fix-point."+
                                        " Switching to retained mode.");
        return false; }
// the system has achieved an evolutionary fix-point
// return:
//                                      true - stay in current mode
//                                      false - switch to retained mode

public AnswerSets TOADs_newOutputReady(int currentcycle, AnswerSets lastOutput) { return lastOutput; }
// a new output has been computed. Expected to return the modified output to be broadcast

public OCLP_CFAS TOADs_CFASReceived(int currentcycle, OCLP_CFAS cfas) { return cfas; }
// a CFAS message was received

public OCLP_message TOADs_OCLPmessageReceived(int currentcycle, OCLP_message OCLPmessage)
{ return OCLPmessage; }
// a reply to a CFAS (i.e. an OCLP_message) was received

public void TOADs_CFAStimeoutExpired(int currentcycle) { }
// CFAS timer expired, will no longer wait for incoming CFAS for this cycle and start the next one



protected void setup() {
        System.out.println("Toads jumping all around! Toad running version Beta 1005.0025");

        // Obtain the parameters. Parameter should technically be a single one, as no spaces are allowed
        Object[] arguments = getArguments();
        String argument = null;
        if (arguments != null) {    // If there are arguments at all
           argument = (String) arguments[0];
           if (arguments.length > 1)
                System.err.println(getLocalName()+": Warning: do not leave blank spaces between parameters."+
                                                " Extra parameters ignored");
           // Attempt to parse the parameters. Write out help and quit if erroneous.
           try{ parseParameters(argument);
           }catch(Exception exce){
                System.err.println(getLocalName()+": incorrect parameter use \n"+prepareUsage());
                doDelete(); }
        }

        // Report MODE
        if (retainedMode) {
           System.out.println("Running in Retained Mode.");
        }else{
           System.out.println("Running in Runtime Mode.");
        }

        language = new SLCodec();
        ontology = ToadsOntology.getInstance();

        ContentManager manager = getContentManager();
        manager.registerLanguage(language);
        manager.registerOntology(ontology);

        TOADs_init();
        // Transfer control to superclass, await for the user's program
        // to be ready to start. Default process is empty.

        // If in runtime mode, start up the cycle
        if (!retainedMode) {
           startNewCycle();
           // This spawns a new MainEngineBehaviour;
        }

        // At the beginning no updating is needed just compute the answer sets of the original OCLP
        lastOutput = OCLPAgentTools.solveOCLP(brain, credulous, getOCT_PATH(), "");
        System.out.println("Output prepared:\n"+OCLPAgentTools.AnswerSets_toString(lastOutput));

        CFASListenerBehaviour CFASlistener =
           new CFASListenerBehaviour(
             this,
             commChannels.getChannelOutput(),
             MessageTemplate.and(MessageTemplate.and(
                        MessageTemplate.MatchOntology(ontology.getName()),
                        MessageTemplate.MatchLanguage(language.getName())),
                MessageTemplate.and(MessageTemplate.MatchProtocol(
                        InteractionProtocol.FIPA_QUERY),
                        MessageTemplate.MatchPerformative(ACLMessage.REQUEST_WHENEVER))));
        addBehaviour(CFASlistener);

        System.out.print(getLocalName()+": started.");


        /**************************************************************/
        if (retainedMode) {
           System.out.println("Running in Retained Mode.");
```

```
        // Add a behaviour to listen for queen messages requiring information from this agent
        FacilitateInfoBehaviour queenListener_INFO =
                new FacilitateInfoBehaviour(
                 this,
                 MessageTemplate.and(MessageTemplate.and(
                                MessageTemplate.MatchOntology(ontology.getName()),
                                MessageTemplate.MatchLanguage(language.getName())),
                        MessageTemplate.and(
                                MessageTemplate.MatchProtocol(InteractionProtocol.FIPA_QUERY),
                                MessageTemplate.MatchPerformative(ACLMessage.REQUEST))));
        addBehaviour(queenListener_INFO);

        // Add a behaviour to listen for queen messages requiring this agent to update its information
        UpdateBehaviour queenListener_UPDATE =
                new UpdateBehaviour(
                 this,
                 MessageTemplate.and(
                      MessageTemplate.and(MessageTemplate.MatchOntology(ontology.getName()),
                                MessageTemplate.MatchProtocol(InteractionProtocol.FIPA_REQUEST)),
                        MessageTemplate.MatchPerformative(ACLMessage.REQUEST)));

        addBehaviour(queenListener_UPDATE);

        // Add a behaviour to listen for queen's STEP/RUN messages
        // These must have protocol FIPA_PROPOSE, performative PROPOSE
        RetainedBehaviour queenListener_STEP =
                new RetainedBehaviour(
                 this,
                 MessageTemplate.and(MessageTemplate.and(
                                MessageTemplate.MatchOntology(ontology.getName()),
                                MessageTemplate.MatchProtocol(InteractionProtocol.FIPA_PROPOSE)),
                        MessageTemplate.MatchPerformative(ACLMessage.PROPOSE)));

        addBehaviour(queenListener_STEP);

        // Add a behaviour to listen for subscription request (by the queen)
        // These must have protocol FIPA_SUBSCRIBE, performative SUBSCRIBE
        SubscriptionHandlerBehaviour queenListener_SUBSCRIBE =
                new SubscriptionHandlerBehaviour(
                 this,
                 theQueen,
                 MessageTemplate.and(MessageTemplate.and(
                                MessageTemplate.MatchOntology(ontology.getName()),
                                MessageTemplate.MatchLanguage(language.getName())),
                        MessageTemplate.and(
                                MessageTemplate.MatchProtocol(InteractionProtocol.FIPA_SUBSCRIBE),
                                 MessageTemplate.MatchPerformative(ACLMessage.SUBSCRIBE))));
        addBehaviour(queenListener_SUBSCRIBE);
    }
}

private String prepareUsage (){
    // Prepares the help text
    String usage = "TOADs' OCLPAgent. commandline use \n \n \t"+
"java -classpath\" .../toads.jar\" jade.Boot [jade_options] AgentNAME:worldpeace.toads.OCLPAgent(options) \n \n"+
     "where options can be one or more of the following, separated by a comma: \n \n"+
     "queen:<queenName> \t forces Retained Mode. <queenName> is the name of the agent's queen. \n"+
     "oct:<oct_path> \t \t overrides oct path to <oct_path>. Default used is ./oct \n"+
     "log:<log_file> \t \t saves a log of the agent's output to the file specified. \n \n";
    return usage;
}

private ACLMessage prepareCallForAnswerSets(jade.util.leap.List agents)
    // Prepares the call for answer sets message to be sent to the channel input agents
{
    OCLP_CFAS cfas = new OCLP_CFAS();
    cfas.setCycle(currentcycle + 1);   // A cycle N prepare CFAS to be replied by agents running cycle N + 1
    ACLMessage CFAS =
        OCLPAgentTools.wrapMessage(this,
                                   null,
                                   cfas,
                                   ACLMessage.REQUEST_WHENEVER,
                                   InteractionProtocol.FIPA_QUERY,
                                        ontology.getName(), language.getName());
    // set deadline to OCLP_message_TIMEOUT milliseconds after now
    CFAS.setReplyByDate(new Date(System.currentTimeMillis() + OCLP_message_TIMEOUT));

    for (int i = 0; i < agents.size(); i++)
      {
        CFAS.addReceiver((AID) agents.get(i));
      }
    return CFAS;
}

private void parseParameters (String parameters)
    // interprets the parameters input and updates the agent accordingly
{
    System.out.println("...parsing parameters: "+parameters);
    parameters+=",";
    String occur = "";
    String option, value;
    String oclprogram = "";
    String aline = "";
    Pattern p = Pattern.compile("[^,]+,");
    Matcher m = p.matcher(parameters);

    while (m.find()){
      occur = m.group();
      occur = occur.substring(0, occur.length() - 1); // take away the ','
      option = occur.split(":", 2)[0];
      value = occur.split(":", 2)[1];
      if (option.equals("queen")) {           // queen:... option
          System.out.println(">> Accepting "+value+" as my Queen!");
          theQueen = new AID(value, AID.ISGUID);
          retainedMode = true;
      } else if (option.equals("oct")) {      // oct:... option
          System.out.println("Alternative OCT path used: "+value);
          setOCT_PATH(value);
      } else if (option.equals("open")) {
          try{
            BufferedReader in
                    = new BufferedReader(new FileReader(value));
            while ((aline = in.readLine()) != null)
                    oclprogram += aline + "\n";
            in.close();
            System.out.println("Read file "+value+". Program is \n"+oclprogram);
            brain = OCLPAgentTools.OCLP_program_fromString(
                    OCLPAgentTools.removeComments(oclprogram) );
            System.out.println("File read and interpreted as: \n"+
                                        OCLPAgentTools.OCLP_program_toString(brain));
          }catch(Exception exce){ System.out.println("Error with file "+value+". "+exce); }
      } else if (option.equals("input")) {
          if (value.indexOf("@") != -1) {          // If it is a remote agent
            System.out.println("Adding the remote agent "+value+" to my valid inputs...");
            commChannels.addChannelInput(new AID(value, AID.ISGUID));
          }else{ // If it is a local agent
            System.out.println("Adding the local agent "+value+" to my valid inputs...");
            commChannels.addChannelInput(new AID(value, AID.ISLOCALNAME));
          }
      } else if (option.equals("output")) {
          if (value.indexOf("@") != -1) { // If it is a remote agent
            System.out.println("Adding the remote agent "+value+" to my valid outputs...");
            commChannels.addChannelOutput(new AID(value, AID.ISGUID));
          }else{ // If it is a local agent
            System.out.println("Adding the local agent "+value+" to my valid outputs...");
            commChannels.addChannelOutput(new AID(value, AID.ISLOCALNAME));
          }
```

```
            } else if (option.equals("semantics")) {
                        if (value.equals("s"))
                           this.setSceptic();
                        else if (value.equals("c"))
                           this.setCredulous();
                        else {          // Invalid semantics
                           System.out.println(prepareUsage()+"Incorrect semantics, found: "+value+
                                                      "; expected c or s. \n");
                           this.doDelete();
                        }
            } else if (option.equals("starter")) {
                        if (value.equals("true"))
                           this.setIsStartingAgent(true);
            }
            // etc etc - more arguments to be implemented
         }
}

public void stopCFASTimer() {
         if (CFAStimer != null) {
            System.out.println("# CFAS timer was terminated before timeout expired.");
            removeBehaviour(CFAStimer);      // If still active, remove the CFAS timer
         }
}

public void startCFASTimer() {
         CFAStimer = new CFASTimerBehaviour(this, CFAS_TIMEOUT);
         addBehaviour(CFAStimer);
}

public void startNewCycle() {
         addBehaviour(new MainEngineBehaviour(this,
                        prepareCallForAnswerSets(getCommChannels().getChannelInput())));
}

public void prepareNewCycle() {
         //Exception exce = new Exception();
         this.setLastEvolutionaryFlag(this.getEvolutionaryFlag());
         this.setEvolutionaryFlag(true);
         if (!this.getRetainedMode()) {
            this.startNewCycle();      // In runtime mode, proceed to the next cycle
         }else{
            this.setCycleCompleted(true);   // In retained mode, wait for the Queen's command to proceed
         }
         this.setNumberCFASreplied(0);            // Reset this counter for the new cycle
}

public ACLMessage prepareSubscriptionReportMessage(AID queenId, OCLP_program updatedprogram)
         // prepares a message to report agent's internal info and puts queedId as receiver
         // the message protocol is sent to FIPA_SUBSCRIBE and the performative SUBSCRIBE
         // the oclp sent is the temporarily updated OCLP to display incoming information
{
         Q_responseMessage info = new Q_responseMessage();
         info.setInternalOCLP(updatedprogram);
         info.setCommChannels(commChannels);
         info.setCredulous(credulous);
         info.setLastOutput(lastOutput);
         info.setEvolutionaryFlag(evolutionaryFlag);
         info.setCycle(currentcycle);
         ACLMessage subsMsg = new ACLMessage(ACLMessage.INFORM);
         subsMsg.setOntology(ontology.getName());
         subsMsg.setLanguage(language.getName());
         subsMsg.setProtocol(InteractionProtocol.FIPA_SUBSCRIBE);
         subsMsg.addReceiver(queenId);         // To the queen...
         Action action = new Action(getAID(), info);
         try{ getContentManager().fillContent(subsMsg,action); }catch(Exception exce){
            System.err.println(getLocalName()+": Error preparing the subscribe message."+
                                    " Please report this bug"); }
         return subsMsg;
}

public ACLMessage prepareBroadcast(Agent myAgent, AnswerSets opinions, ACLMessage cfas,
                                                      int cycle, boolean evolutionaryFlag)
         // Function responsible of wrapping the AnswerSets object and preparing
         // the message to be boradcast as a reply message to "cfas"
{
         ACLMessage msg = cfas.createReply();
         msg.setProtocol(InteractionProtocol.FIPA_PROPOSE);          // FIPA PROPOSE protocol is used

         OCLP_message OCLPmessage = new OCLP_message();
         OCLPmessage.setAnswerSets(opinions);
         OCLPmessage.setEvolutionaryFlag(evolutionaryFlag);

         Action action = new Action(myAgent.getAID(), OCLPmessage);

         try{
            myAgent.getContentManager().fillContent(msg,action);
            System.out.println(myAgent.getLocalName() + ": preparing to broadcast message: "+
                                    OCLPAgentTools.AnswerSets_toString(opinions));
         }catch(OntologyException e){
            System.err.println(myAgent.getLocalName()+":Error with ontology @863OA in request:" + e);
            return null;
         }catch(Codec.CodecException e){
            System.err.println(myAgent.getLocalName()+":Error with codec @866OA in request:" + e);
            return null;
         }
         msg.setPerformative(ACLMessage.INFORM);
         return msg;
}

}
```

B.2 CFASListenerBehaviour.java

```
package worldpeace.toads;

import jade.proto.AchieveREResponder;
import jade.util.leap.*;

import jade.content.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import jade.content.lang.*;
import jade.content.lang.sl.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import toads.ontology.*;

public class CFASListenerBehaviour extends AchieveREResponder {
   // Listens for CFAS (Call for Answer Sets). If an answer is ready for them, it is sent,
   // if not they are recorded in CFSs
   // Agents with no valid inputs will respond to CFAS immediately with their ethernal answer sets
   // Agents catalogued as "Starting agents" are required to reply to CFAS for a cycle ahead

   ACLMessage response;
   ACLMessage outputMsg;
   OCLP_CFAS cfas = null;
   OCLPAgent myAgent;
   jade.util.leap.List outputs;
   int startingAgentShift = 0; // Used to handle starting agents

   public CFASListenerBehaviour(OCLPAgent a, jade.util.leap.List validoutput, MessageTemplate mt) {
            super(a, mt);
            myAgent = a;
            outputs = validoutput;
   }

   public ACLMessage prepareResponse(ACLMessage msg) {
            // if CFAS is coming from an agent who is not a valid output, ignore it
            if (outputs.indexOf(msg.getSender()) == -1) {
               System.out.println("Received a CFAS, but it isn't from one of my valid outputs! Ignoring it.");
               return null;
            }

            response = msg.createReply();

            try{ cfas = (OCLP_CFAS) ((Action) myAgent.getContentManager().extractContent(msg)).getAction(); }
            catch(Exception exce){
               System.err.println(myAgent.getName()+
                                  ": error trying to identify a CFAS from"+
                                  msg.getSender().getName()+
                                  ". Possibliy a class cast error. Please report the bug.");
               exce.printStackTrace(); return null; }

            // Fire appropriate event and update the cfas - default event just returns the second parameter
            cfas = myAgent.TOADs_CFASReceived(myAgent.getCurrentcycle(), cfas);

            // Starting agents will respond to CFAS's from a cycle ahead
            if (myAgent.getIsStartingAgent())
               startingAgentShift = 1;

            if (((cfas.getCycle() != myAgent.getCurrentcycle() + startingAgentShift) &&
                 !myAgent.getCommChannels().getChannelInput().isEmpty()) ||
               (!myAgent.getCycleCompleted())) {
               // CFAS corresponding to other cycles get stored, same if output isn't ready
               // However, input agents (agent with no input channels) don't have a cycle
               System.out.println("Received a CFAS message from "+msg.getSender().getName()+" Cycle:"+
                                  cfas.getCycle()+". Storing CFAS.");
               String reason;
               reason = "";
               if (cfas.getCycle() != myAgent.getCurrentcycle())
                        reason = "CFAS cycle was: "+cfas.getCycle()+". CurrentCycle is: "+
                           myAgent.getCurrentcycle();
               if (!myAgent.getCycleCompleted())
                        reason += " & cycle is not yet completed";
               System.out.println("Reason: "+reason);
               myAgent.getCFASs().add(msg);
               return null;
            }else{                                          // if output ready, respond with it
               System.out.println("Received a CFAS message from "+msg.getSender().getName()+
                                  " Cycle:"+cfas.getCycle()+
                                  ". Reply is \n "+
                                  OCLPAgentTools.AnswerSets_toString(myAgent.getLastOutput()));
               outputMsg =          myAgent.prepareBroadcast(myAgent,
                                                             myAgent.getLastOutput(),
                                                             msg, myAgent.getCurrentcycle(),
                                                             myAgent.getEvolutionaryFlag());
               myAgent.getLastOutputMessages().add(outputMsg);
               response = outputMsg;
               myAgent.setNumberCFASreplied(myAgent.getNumberCFASreplied() + 1);
            }

            if (myAgent.getNumberCFASreplied() == myAgent.getCommChannels().getChannelOutput().size()) {
               myAgent.stopCFASTimer();
               myAgent.prepareNewCycle();       // Reset variables and prepare for the new cycle
            }

            return response;
         }
   };
```

B.3 MainEngineBehaviour.java

```
package worldpeace.toads;

import jade.util.leap.*;
import jade.proto.AchieveREInitiator;
import jade.content.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import jade.content.lang.*;
import jade.content.lang.sl.*;

import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
import toads.ontology.*;

public class MainEngineBehaviour extends AchieveREInitiator {
   private boolean finished = false;
   private AnswerSets solutions = new AnswerSets();
   ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
   OCLPAgent myAgent;

   public MainEngineBehaviour(OCLPAgent agent, ACLMessage initiator){
            super(agent, initiator);
            myAgent = agent;
   }

   public void setup() {
            System.out.println("Initialising cycle "+myAgent.getCurrentcycle());
            if (myAgent.getCommChannels().getChannelInput().isEmpty()) {
               myAgent.setCurrentcycle(myAgent.getCurrentcycle() + 1);    // Only update cycle counter
               return;                        // Input agent will not send CFAS's and therefore will have an output ready always
            }
            myAgent.setCycleCompleted(false);   // Cycle starting, will switch back to true when all messages have been sent
            myAgent.setLastOutput(null);            // The new output will not be ready until it's recalculated
            myAgent.setEvolutionaryFlag(true);     // Needed to keep track of all incoming evolutionary status information
   }

   protected void handleOutOfSequence(ACLMessage msg) {
            System.out.println("Received an OCLP_message, but it is out of sequence. Ignoring. Perhaps you should extend your
deadlines");
   }

   public void handleInform(ACLMessage msg) {
            int posi, nega;

            OCLP_message OCLPmessage = extractOCLPmessage(msg);

            // Fire appropriate event and update OCLPmessage - default event just returns the second parameter
            OCLPmessage = myAgent.TOADs_OCLPmessageReceived(myAgent.getCurrentcycle(), OCLPmessage);

            System.out.print("Received a proper OCLP_message message, content: " +
OCLPAgentTools.AnswerSets_toString(OCLPmessage.getAnswerSets()));

            // Programmed to use only the first Answer Set of the list of incoming Answer Sets by each agent
            String anAtom;

            if (!OCLPmessage.getAnswerSets().getList().isEmpty()) {

   // Add the incoming answer set to the merged answer set in the agent's field .input (accessed through getInput(), setInput())
               for (posi = 0; posi < ((AnswerSet) OCLPmessage.getAnswerSets().getList().get(0)).getPositive().size(); posi++) {
                        // Add it, but only if it is not already there
                        anAtom = (String) ((AnswerSet) OCLPmessage.getAnswerSets().getList().get(0)).getPositive().get(posi);
                        if (myAgent.getInput().getPositive().indexOf(anAtom) == -1)
                           myAgent.getInput().addPositive(anAtom);
               }
               for (nega = 0; nega < ((AnswerSet) OCLPmessage.getAnswerSets().getList().get(0)).getNegative().size(); nega++) {
                        // Add it, but only if it is not already there
                        anAtom = (String) ((AnswerSet) OCLPmessage.getAnswerSets().getList().get(0)).getNegative().get(nega);
                        if (myAgent.getInput().getNegative().indexOf(anAtom) == -1)
                           myAgent.getInput().addNegative(anAtom);
               }

            }

    // If any of the incoming messages does not have the evolutionaryFlag active, then the system cannot be stable... yet
            myAgent.setEvolutionaryFlag( myAgent.getEvolutionaryFlag() && OCLPmessage.getEvolutionaryFlag() );

   }

   public void handleAllResponses(java.util.Vector responses)
   {
            int contra, i;                  // dummy counters
            ACLMessage outputMsg = null;
            OCLP_program updatedOCLP = null;
            int startingAgentShift = 0;

            // Filter the input
            System.out.println("All messages received. Proceeding to filter the input");

            // Obtain a list of contradictory atoms, i.e. atoms present in both the positive and the negative part
            List contradictory = OCLPAgentTools.generateContradictoryList(myAgent.getInput());

 //System.out.println("**** Contradictory list prepared. Total of " + contradictory.size() + " contradictory atoms found");

            // Scan positive and negative part to remove items which are in 'contradictory'
            for (contra = 0; contra < contradictory.size(); contra++)
               {
                        while (myAgent.getInput().getPositive().contains(contradictory.get(contra)))
                           myAgent.getInput().removePositive((String) contradictory.get(contra));
                        while (myAgent.getInput().getNegative().contains(contradictory.get(contra)))
                           myAgent.getInput().removeNegative((String) contradictory.get(contra));
               }

//System.out.println("Attempting to update OCLP brain. Previous was: \n" +
OCLPAgentTools.OCLP_program_toString(myAgent.getOCLP()));
            //System.out.println("------");

            // Testing program updating
            updatedOCLP = OCLPAgentTools.upDateOCLP(OCLPAgentTools.copyOCLP_program(myAgent.getOCLP()),
myAgent.getInput(),
                                                                                    "A"+myAgent.getCurrentcycle());

            // System.out.println("Updated OCLP is: \n" + OCLPAgentTools.OCLP_program_toString(updatedOCLP)+"\n");

            myAgent.getInput().clearAllPositive(); myAgent.getInput().clearAllNegative();          // clear the buffered input

            // Call oct with defaut parameters and default path "./oct"
            solutions = OCLPAgentTools.solveOCLP(updatedOCLP, myAgent.getCredulous(), myAgent.getOCT_PATH(), "");

            // Fire appropriate event and update solutions - default event just returns the second parameter
            solutions = myAgent.TOADs_newOutputReady(myAgent.getCurrentcycle(), solutions);

            System.out.println("Solutions for this cycle are: \n"+OCLPAgentTools.AnswerSets_toString(solutions)+"\n");

            if (myAgent.getEvolutionaryFlag() == true && OCLPAgentTools.AnswerSets_compare(solutions,
myAgent.getLastOutput()))
               myAgent.setEvolutionaryFlag(true);
            else
               myAgent.setEvolutionaryFlag(false);

            // React to the evolutionary fix-point status: stop execution,
            // fire events and -if in retainedMode- send message to the queen.
            if (myAgent.getEvolutionaryFlag() && myAgent.getLastEvolutionaryFlag())
               {
```

```
                    if (!myAgent.TOADs_achievedEvolutionaryFixPoint(myAgent.getCurrentcycle(),
myAgent.getLastOutput()))          // fire appropriate event
                        myAgent.setRetainedMode(true);    // if not asked otherwise, the agent will switch to Retained Mode
                }

                myAgent.setLastEvolutionaryFlag(myAgent.getEvolutionaryFlag());
                myAgent.setLastOutput(solutions);
                myAgent.setNumberCFASreplied(0);
                myAgent.setCurrentcycle((myAgent.getCurrentcycle()) + 1);     // new solutions calculated -> new cycle running

                // Scan through all pending CFAS's for this cycle
                OCLP_CFAS cfas = null;
                ACLMessage cfasMsg = null;

                // Starting Agents will respond to CFAS corresponding to 1 cycle ahead
                if (myAgent.getIsStartingAgent())
                   startingAgentShift = 1;
                else
                   startingAgentShift = 0;

                for (i = 0; i < myAgent.getCFASs().size(); i++) {
                   cfasMsg = (ACLMessage) myAgent.getCFASs().get(i);
                   try{
                        cfas = (OCLP_CFAS) ((Action) myAgent.getContentManager().extractContent(cfasMsg)).getAction();
                        if (cfas.getCycle() == myAgent.getCurrentcycle() + startingAgentShift) {
                           myAgent.setNumberCFASreplied(myAgent.getNumberCFASreplied() + 1);
                           System.out.println("For cycle: "+myAgent.getCurrentcycle()+                                    }
                                   "Sending reply (OCLP_message) to pending CFAS, and removing it from list.");          };
                           outputMsg = myAgent.prepareBroadcast(myAgent, myAgent.getLastOutput(), cfasMsg,
                                                                   myAgent.getCurrentcycle(),
myAgent.getEvolutionaryFlag());
                           myAgent.getLastOutputMessages().add(outputMsg);
                           // send the reply
                           myAgent.send(outputMsg);
                           myAgent.getCFASs().remove(i);     // And remove the CFAS from the list of pending CFAS's
                        } else {
                           //  System.out.println("Future CFAS scanned, corresp. to cycle "+cfas.getCycle()+"; from agent"+
                           //                  msg.getSender().getLocalName());
                        }
                   }catch(Exception exce){ //System.err.println(myAgent.getName()+": error trying to identify a CFAS from"+
                        //msg.getSender().getName()+". Possibliy a class cast error. Please report the bug.");
                        exce.printStackTrace(); }
                }

                if (myAgent.getSubscription()){
                   // If we are subscribed, we have to send an additional message to the queen.
                   myAgent.send(myAgent.prepareSubscriptionReportMessage(myAgent.getTheQueen(), updatedOCLP));
                }

                // Once the output is ready, start the timer for CFAS to arrive
                // output agents will have X milliseconds to send their CFAS, after that time
                // the cycle will finish anyway
                myAgent.startCFASTimer();
        }

        public OCLP_message extractOCLPmessage(ACLMessage msg)
                // A function to extract the ontology
        {
                /******************************************************** Proper object wrapping */
                ContentElement content = null;
                OCLP_message OCLPmessage = new OCLP_message();

                try{
                   try{
                        content = myAgent.getContentManager().extractContent(msg);
                   }catch(OntologyException e){
                        System.err.println(myAgent.getLocalName()+":Error relating to message and " +
                                                "ontology:" + e);
                        e.printStackTrace();
                        myAgent.send(OCLPAgentTools.createErrorReply(msg));
                        return null;
                   }catch(Codec.CodecException e){
                        System.err.println(myAgent.getLocalName()+":Error Parsing the message format:"
                                                + e);
                        e.printStackTrace();
                        myAgent.send(OCLPAgentTools.createErrorReply(msg));
                        return null;
                   }

                }catch(Exception e){

                   System.err.println(myAgent.getLocalName() + ": Sending not-understood");
                   myAgent.send(OCLPAgentTools.createErrorReply(msg));
                   return null;
                }

                try{ OCLPmessage = (OCLP_message) ((Action)content).getAction(); }
                catch(Exception e){
                   System.err.println(myAgent.getLocalName()+": error casting OCLP_message <- ContentElement "+e);
                   System.err.println("---------- object 'content' was "+content);
                   return null;
                }

                return OCLPmessage;
```

B.4 QueenAgent.java

```
package worldpeace.toads;

import jade.core.*;
import jade.core.behaviours.*;
import jade.proto.SimpleAchieveREInitiator;
import jade.proto.AchieveREInitiator;
import jade.lang.acl.*;
import toads.ontology.*;
// import java.util.*;
import java.util.Vector;
import java.io.*;
import java.util.Calendar;
import jade.util.BasicProperties;
import jade.util.leap.*;

import jade.content.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import jade.content.lang.*;
import jade.content.lang.sl.*;
import jade.wrapper.AgentController;

import jade.domain.FIPANames.InteractionProtocol;

import worldpeace.toads.OCLPAgentTools;

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.TreeSelectionListener;
import javax.swing.event.TreeSelectionEvent;


import worldpeace.toads.OCLPAgentTools;          // Various tools for TOADs
import worldpeace.toads.Queen;                   // Queen automatically generated GUI code
import worldpeace.toads.DynamicTree;             // Dynamic tree functions
import worldpeace.toads.AgentSpec2; // simple extension of AgentSpec to allow .toString operation
import worldpeace.toads.AgentWatcherGUI;
import worldpeace.toads.RetrieveInformationBehaviour;


public class QueenAgent extends Agent{
   Codec language;
   Ontology ontology;
   jade.util.leap.List agentList = new jade.util.leap.ArrayList();
   private QueenInterface Q = null;
   public OCLPAS_system OCLPAS = new OCLPAS_system();          // Information about the whole system.

   public QueenAgent() {
            super();
   }

   protected void setup() {
            language = new SLCodec();
            ontology = ToadsOntology.getInstance();

            ContentManager manager = getContentManager();
            manager.registerLanguage(language);
            manager.registerOntology(ontology);

     Frame window = new Frame("TOAD Queen 0.1");

     window.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                 System.exit(0);
                        }
            });

            Q = new QueenInterface(this);
   Q.init();
   window.add("Center", Q);
   window.pack();
   window.setVisible(true);

            // Need mouselistener to listen for doubleclicks
            Q.tree.getTree().addMouseListener(new MyMouseListener(this));

            }

            public QueenInterface getQueenInterface() {return Q;}
            public OCLPAS_system getOCLPAS() {return OCLPAS;}

public class SendStepBehaviour extends AchieveREInitiator {
            // Sends a STEP/RUN message and waits for confirmation from agents

            public SendStepBehaviour(Agent agent, ACLMessage msg) {
               super(agent, msg);
            }

            protected void handleNotUnderstood(ACLMessage msg) {
               System.out.println("Agent "+msg.getSender().getName()+
                        " has not understood the RUN/STEP command. Please check the agent is running properly.");
               return;
            }

            protected void handleRefuse(ACLMessage msg) {
               System.out.println("Agent "+msg.getSender().getName()+" has refused to follow the RUN/STEP "+
                                    "command. Agent might be busy. Please check the agent is running properly.");
               return;
            }

            protected void handleAllResultNotifications(java.util.Vector resultNotifications) { return; }

            protected void handleAllResponses(java.util.Vector responses) {          return; }


};

public class SubscriberListenerBehaviour extends SimpleBehaviour {
            // On creation it sends a message to the AID passed as a parameter initiating a SUBSCRIBE protocol
            // It will listen for incoming messages under the subscription protocol so as to update
            // the window for this agent
            // whose handler has to be passed as an argument as well.
            AID receiver;
            AgentEdit myDialog;
            Agent myAgent;
            boolean finished = false;
            ACLMessage msg = null;
            int i = 0;      // dummy counter

            public SubscriberListenerBehaviour (Agent agent, AID rec, AgentEdit ae) {
               super(agent);
               myAgent = agent;
               receiver = rec;
               myDialog = ae;
               // Wrap and send a setup subscription message
               myAgent.send(OCLPAgentTools.wrapMessage(myAgent,
                                                                  receiver,
                                                                  (new Q_setupSubscribe()),
                                                                  ACLMessage.SUBSCRIBE,
                                                                  InteractionProtocol.FIPA_SUBSCRIBE,
                                                                  ontology.getName(), language.getName()));
            }
```

```
        public void setup() {
            return;
        }

        public void action() {
            // Listen for msgs matching ontology, language;
            // with protocol FIPA_SUBSCRIBE and performative INFORM.
            block();
            msg = receive(MessageTemplate.and(
                        MessageTemplate.and(MessageTemplate.MatchOntology(ontology.getName()),
                                                    MessageTemplate.MatchSender(receiver)),
                        MessageTemplate.and( MessageTemplate.MatchProtocol(InteractionProtocol.FIPA_SUBSCRIBE),
                                                    MessageTemplate.MatchPerformative(ACLMessage.INFORM))));
            if (msg == null)
                        return;
            ACLMessage answer = msg.createReply();
            Q_responseMessage command;
            try{ command = (Q_responseMessage) ((jade.content.onto.basic.Action)

        myAgent.getContentManager().extractContent(msg)).getAction(); }
            catch(Exception exce){
                        System.err.println(myAgent.getName()+
                          ": error trying to identify an INFORM message under the SUBSCRIPTION protocol.\n"+
                                                "Possibliy a class cast error. Please report the bug."); return; }

            System.out.println("Received information for agent "+receiver.getName()+
                                    ". \n *** Changes should appear in the GUI now.");
            // Display OCLP program
            myDialog.text_OCLP.setText(OCLPAgentTools.OCLP_program_toString(command.getInternalOCLP()));
            // Add a line stating the incoming event and some properties - cycle & evol. flag
            Calendar rightNow = Calendar.getInstance();
            String mytime = ""+rightNow.get(Calendar.HOUR_OF_DAY)+":"+rightNow.get(Calendar.MINUTE)+
                        ":"+rightNow.get(Calendar.SECOND);
            myDialog.agentConsole.setText(myDialog.agentConsole.getText()+"\n"+
                                                            mytime+" > Cycle "+command.getCycle()+
                                                            " completed. Evolutioary flag was "+
                                                            command.getEvolutionaryFlag());

            // Display the last output
            myDialog.lastOutputText.setText(OCLPAgentTools.AnswerSets_toString(command.getLastOutput()));
            // Update the title of the window to reflect the current cycle and evolutionary status
            String theStatus = "UNSTABLE";
            if (command.getEvolutionaryFlag())
                        theStatus = "STABLE 2 cycles in a row";
            if (command.getFixpoint())
                        theStatus = "FIX - POINT";
            myDialog.setTitle(produceAgentName(receiver)+" - Cycle: "+command.getCycle()+
                                    " | Evolutionary status: "+theStatus);
            return;
        }

        public boolean done() { return finished; }
        public void stop() {finished = true; return; }
};

class MyMouseListener implements MouseListener {
        // This listener's purpose is to react to double-clicks, opening the agent edit window

        QueenAgent myAgent;
        AgentSpec2 agentSpec;
        AgentSpec2 auxSpec;

        public MyMouseListener(QueenAgent agent) { super(); myAgent = agent; }

        public void mouseReleased(MouseEvent mouse) { return; }
        public void mousePressed(MouseEvent mouse) { return; }
        public void mouseExited(MouseEvent mouse) { return; }
        public void mouseEntered(MouseEvent mouse) { return; }

public void mouseClicked(MouseEvent mouse) {
    TreePath currentSelection;
    DefaultMutableTreeNode node;
    int i = 0; // dummy counter

    System.out.println("Mouse clicked.");

    if ((mouse.getButton() == mouse.BUTTON3) && (mouse.getClickCount() >= 2)){
                System.out.println("Mouse RightClicked.");
                currentSelection = Q.tree.getTree().getPathForLocation(mouse.getX(), mouse.getY()) ;
                System.out.println("Path for location: "+currentSelection);
                if (currentSelection != null) {
                    node = (DefaultMutableTreeNode)
                                (currentSelection.getLastPathComponent());
                    System.out.println("Node here is: "+node);

                    try{ agentSpec = (AgentSpec2) node.getUserObject(); }
                    catch(Exception exce) {
                                System.err.println("Invalid selection. Please select an agent from the list.");
                                return; }
                    if (agentSpec.getIsBeingMonitored()) {
                                System.out.println("Sorry. Agent is already being monitored.");
                    }
                    agentSpec.setIsBeingMonitored(true);

                    // Create the Edit Agent Dialog object
                    // This needs information about the agent being monitored
                    AgentWatcherGUI AgentEditDialog = new AgentWatcherGUI(Q._getFrame(Q), agentSpec, myAgent);

                    AgentEditDialog.setTitle(node.toString());
                    //AgentEditDialog.text_OCLP.setText(Q.text_OCLP.getText());
                    AgentEditDialog.text_OCLP.setPreferredSize(new Dimension(1000, 1000));

                    // Fill in the list of outputs
                    DefaultListModel listModel = new DefaultListModel();
                    AgentEditDialog.OutputAIDs.setModel((ListModel) listModel);
                    listModel.clear();
                    for (i = 0; i < agentSpec.getChannelOutput().size(); i++){
                                listModel.addElement( produceAgentName((AID) agentSpec.getChannelOutput().get(i)) );
                                // We use the ListModel to add the item because we ought to be able to access it through
                                // the ListModel later on, when we want to remove items from the list
                    }

                    // Fill in the combo box
                    System.out.println("Trying to fill the combo box, "+agentList.size()+" items to add.");
                    listModel = new DefaultListModel();
                    AgentEditDialog.selectOutput.setModel((ListModel) listModel);
                    listModel.clear();
                    for (i = 0; i < agentList.size(); i++){
                                if ( !((AID) agentList.get(i)).equals(agentSpec.getId()) )
                                    listModel.addElement( produceAgentName((AID) agentList.get(i)) );
                    }
                    // STEP 1: add a listener behaviour to receive information of this agent in real time.

                    SubscriberListenerBehaviour mySubscribedBehaviour =
                                new SubscriberListenerBehaviour(myAgent, agentSpec.getId(), AgentEditDialog);

                    addBehaviour(mySubscribedBehaviour);

                    // STEP 2: Create a listener to kill the behaviour and unsubscribe the agent when
                    // the dialog is closed. Create also a listener to spawn a requireInformationBehvaiour
                    // when the "Download" button is pushed
                    // And one that uploads information when the button "Upload" is pushed
                    // And another two to update agentSpec upon addition/removal of valid output agents
                    AgentEditDialog.addWindowListener(new TOAD_WC_WindowAdapter(agentSpec,
                                                                                myAgent,
```

```
mySubscribedBehaviour){
        public void windowClosing(WindowEvent e) {
            System.out.println("Window closed, sending Q_cancelSubscrption to agent "+
                agentSpec.getId().getName());
                agentSpec.setIsBeingMonitored(false);
                myAgent.send(OCLPAgentTools.wrapMessage(myAgent,
                agentSpec.getId(),
                (new Q_cancelSubscribe()),
                ACLMessage.SUBSCRIBE,
                InteractionProtocol.FIPA_SUBSCRIBE,
                ontology.getName(),
                language.getName()));
                mySubscribedBehaviour.stop();          // halt the behaviour, stop listening
              }
            });


          // STEP 3: Show the window
          AgentEditDialog.show();
        }
        return;
    }
    else if (mouse.getButton() == mouse.BUTTON1) {
        currentSelection = Q.tree.getTree().getPathForLocation(mouse.getX(), mouse.getY()) ;
        if (currentSelection != null) {
          node = (DefaultMutableTreeNode)
              (currentSelection.getLastPathComponent());

          if (node == null) return;

          if (node.toString().equals("Valid INPUT"))
              {
              }
          else if (node.toString().equals("Valid OUTPUT"))
              {
              }
          else if (node.toString().equals("Agent List"))
              {
              }
          else{ // if an agent is selected, fill the right form with its information
              try{ agentSpec = (AgentSpec2) node.getUserObject(); }
              catch(Exception exce){
                 System.err.println("Error, target selected is not an agent, but it should be!"+
                                    " Class was: "+node.getUserObject().getClass()); return;
              }
              addBehaviour(new RetrieveInformationBehaviour(myAgent,
                                    prepareIRmessage(agentSpec.getId()),
                                        null));
          }

        }
    }
}

class TOAD_WC_WindowAdapter extends WindowAdapter{
    // A special window adapter class to interface with some JADE agent objects

    AgentSpec2 agentSpec;
    Agent myAgent;
    SubscriberListenerBehaviour mySubscribedBehaviour;

    public TOAD_WC_WindowAdapter(AgentSpec2 spec, Agent agent,
                                    SubscriberListenerBehaviour subsBehaviour) {
        agentSpec = spec;
        myAgent = agent;
        mySubscribedBehaviour = subsBehaviour;
    }
};
};
```

```
public void out (String text) {
        // writes some text to the console
        String old = Q.console.getText();
        // Allow a maximum of 10000 characters in the output
        if (old.length() > 10000)
           old = old.substring(old.length() - 10000);
        Q.console.setText(old+text);
}

public DefaultMutableTreeNode findNode (AID id) {
        // Given an agent AID, attempts to find a node in the GUI tree that holds the agentSpec for that node
        DefaultMutableTreeNode theNode;
        DefaultMutableTreeNode root;
        DefaultTreeModel theModel = Q.tree.getDTModel();
        AgentSpec2 theSpec = null;

        for (int i = 0; i < theModel.getChildCount(theModel.getRoot()); i++) {
          theNode = (DefaultMutableTreeNode) theModel.getChild(Q.tree.getRootNode(), i);
          try{ theSpec = (AgentSpec2) theNode.getUserObject(); }
          catch(Exception exce){
                System.err.println("Error finding node, horrible!, node was: "+theNode); return null;
          }
          if (theSpec.getId().equals(id))
                return theNode;
        }
        return null;
}

public AID selectedAgentAID() {
        // returns the AID of the agent selected in the tree, or null if none
        AID myAID = null;
        TreePath currentSelection = Q.tree.getTree().getSelectionPath() ;
        if (currentSelection != null) {
          DefaultMutableTreeNode node = (DefaultMutableTreeNode)
                (currentSelection.getLastPathComponent());
          try{ myAID = ((AgentSpec2) node.getUserObject()).getId();
          }catch(Exception exce){ return null;}
        }
        return myAID;
}

public String produceAgentName(AID id) {
        // Produces a String name for a given AID, to be used as nickname and hence displayed in the tree
        if (Q.fullNames.isSelected())
          return (new String(id.getName()));
        else
          return (new String(id.getLocalName()));
}

public Q_responseMessage extractQ_responseMessage(ACLMessage msg) {
        // A function to extract the ontology object Q_responseMessage
        // this object contains the answer from an agent stating current state information
        /************************************************** Proper object wrapping */
        ContentElement content = null;
        Q_responseMessage QRmessage = new Q_responseMessage();

        try{
          try{
                content = getContentManager().extractContent(msg);
          }catch(OntologyException e){
                System.err.println(getLocalName()+":Error relating to message and " +
                                        "ontology:" + e);
                e.printStackTrace();
                send(OCLPAgentTools.createErrorReply(msg));
                return null;
          }catch(Codec.CodecException e){
                System.err.println(getLocalName()+":Error Parsing the message format:"
                                        + e);
```

```
                e.printStackTrace();
                send(OCLPAgentTools.createErrorReply(msg));
                return null;
            }

        }catch(Exception e){

            System.err.println(getLocalName() + ": Sending not-understood");
            send(OCLPAgentTools.createErrorReply(msg));
            return null;
        }

        try{ QRmessage = (Q_responseMessage) ((jade.content.onto.basic.Action)content).getAction(); }
        catch(Exception e){
            System.err.println(getLocalName()+": error casting Q_responseMessage <- ContentElement "+e);
            System.err.println("---------- object 'content' was "+content);
            return null;
        }

        return QRmessage;
}

public ACLMessage prepareIRmessage(AID id) {
        // prepares a QUERY message to perform an Information request
        ACLMessage inquiry = new ACLMessage(ACLMessage.REQUEST);
        inquiry.setOntology(ontology.getName());
        inquiry.setLanguage(language.getName());
        inquiry.setProtocol(InteractionProtocol.FIPA_QUERY);
        inquiry.setPerformative(ACLMessage.REQUEST);
        inquiry.addReceiver(id);
        return inquiry;
}

public int findOCLPASindex(OCLPAS_system oclpas, AID id) {
        // Finds the index of a given agent (specified by the id) in the internal database
        int result = -1;
        int i = 0; // dummy counter
        for (i = 0; i < oclpas.getAgentsSpecification().size(); i++){
            if (id.equals(((Q_UpdateMessage) oclpas.getAgentsSpecification().get(i)).getToAgent()))
                result = i;
        }
        return result;
}

public void sendUpdateMessage(Q_UpdateMessage updateMsg) {
        // wraps a Q_UpdateMessage into a ACLMessage and sends it
        send(OCLPAgentTools.wrapMessage(this,
                                        updateMsg.getToAgent(),
                                        updateMsg,
                                        ACLMessage.REQUEST,
                                        InteractionProtocol.FIPA_REQUEST,
                                        ontology.getName(),
                                        language.getName()));
}

public AID createNewLocalOCLPAgent(String nickname, AID queensAID, int portNumber, String octpath) {
        // Creates a new containter and adds an OCLP agent with standard properties to it
        AID newAID = null;
        AgentThread parallelThread;
        try{
            ProfileImpl p = new ProfileImpl();
            p.setParameter(Profile.MAIN_PORT, ""+portNumber);
            p.setParameter(Profile.MAIN, "false");

            Object[] args = new Object[1];
            args[0] = "queen:"+queensAID.getName()+",oct:"+octpath;

            newAID = new AID(nickname, AID.ISLOCALNAME);        // Agent is created locally
            jade.wrapper.AgentContainer ac =
                        (jade.wrapper.AgentContainer) jade.core.Runtime.instance().createAgentContainer(p);
            AgentController aCTRL = ac.createNewAgent(nickname, "worldpeace.toads.OCLPAgent", args);
            aCTRL.start();

        }catch(Exception exce){
            System.err.println("Error trying to initialize new agent: "+nickname+" for the queen "+
                                queensAID.getName()+". \n"+exce); }
        return newAID;
}

class AgentThread extends Thread {
        String nickname;
        int portNumber;
        AID queensAID;

        public AgentThread(String name, AID qid, int port) {
            nickname = name;        portNumber = port;          queensAID = qid;
        }

        public void run() { try{
            System.out.println("Executing command line: "+"java jade.Boot -port "+portNumber+
                                " -container "+nickname+":worldpeace.toads.OCLPAgent(queen="+
                                queensAID.getName()+")");
            Process p = java.lang.Runtime.getRuntime().exec("java jade.Boot -port "+portNumber+
                                                                " -container "+nickname+
":worldpeace.toads.OCLPAgent(queen="+
                                                                queensAID.getName()+")");
            BufferedReader stdInput = new BufferedReader(new
                                                                InputStreamReader(p.getInputStream()));
            BufferedReader stdError = new BufferedReader(new
                                                                InputStreamReader(p.getErrorStream()));
            String s = "";
            while ( (s = stdInput.readLine()) != null )
                    System.out.println(">>> "+nickname+" >>>"+s);
            // read any errors from the agent output
            while ((s = stdError.readLine()) != null)
                    System.out.println(">>> "+nickname+" >>[ERROR]>"+s);
        }catch(Exception exce){ System.err.println("Error spawning agent: "+nickname+" for the queen "+
                                                            queensAID.getName()+". \n"+exce); }
        }
};


public class QueenInterface extends Queen {
        QueenAgent myAgent;
        AgentSpec2 agentSpec = null;
        DefaultMutableTreeNode agentNode = null;
        DynamicTree tree;
        int i = 0; //dummy counter
        Q_UpdateMessage newSpec;
        AID receiver;

        public QueenInterface(QueenAgent agent) {
            super();
            myAgent = agent;
            tree = new DynamicTree();
            jSplitPane27.setRightComponent(tree);
        }

        public void actionPerformed(ActionEvent evt) {

            if ( evt.getSource() == jButton44 ) {
                    AddAgent  addAgentDialog = new AddAgent(_getFrame(this));
                    addAgentDialog.show();
                    if (addAgentDialog.byLocalName.isSelected()) {
                        agentSpec = new AgentSpec2();
                        agentSpec.setNickname(addAgentDialog.localName.getText());
                        agentSpec.setId(new AID( addAgentDialog.localName.getText() , AID.ISLOCALNAME));
```

```
            agentNode = tree.addObject(null, agentSpec, true);
            tree.addObject(agentNode, "Valid INPUT", false);
            tree.addObject(agentNode, "Valid OUTPUT", false);
        }
        else if (addAgentDialog.byGUID.isSelected()) {
            agentSpec = new AgentSpec2();
            AID newId;
            newId = new AID(addAgentDialog.GUID.getText() , AID.ISLOCALNAME);
            agentSpec.setNickname(newId.getLocalName());
            agentSpec.setId(newId);
            agentNode = tree.addObject(null, agentSpec, true);
            tree.addObject(agentNode, "Valid INPUT", false);
            tree.addObject(agentNode, "Valid OUTPUT", false);
            // retrieve/produce a short name for an external agent under this GUID - to be done
            // uses addAgentDialog.GUID.getText()
        }
        else if (addAgentDialog.spawnNew.isSelected()){
            // Create a new agent and add it to list - to be done
            // uses class name addAgentDialog.className.getText()
            //  and agent name addAgentDialog.agentName.getText()
            agentSpec = new AgentSpec2();
            agentSpec.setNickname(addAgentDialog.agentName.getText());
            String octpath = "./oct"; // Optional oct path
            octpath = ((String)JOptionPane.showInputDialog(Q,
                                        "Please enter OCT commandline: ",
                                                "Spawning new agents...",
                                                        JOptionPane.PLAIN_MESSAGE,
                                                                null,
                                                                null,
                                                                "./oct") );

            agentSpec.setId(createNewLocalOCLPAgent(agentSpec.getNickname(),
                                                                myAgent.getAID(),
                                                                1234,
                                                                octpath));
            if (agentSpec.getId() == null)
                    return;
            agentNode = tree.addObject(null, agentSpec, true);
            tree.addObject(agentNode, "Valid INPUT", false);
            tree.addObject(agentNode, "Valid OUTPUT", false);
        }
        if(agentSpec != null) {
            newSpec = new Q_UpdateMessage();
            newSpec.setToAgent(agentSpec.getId());
            newSpec.setNickname(agentSpec.getNickname());
            OCLPAS.addAgentsSpecification(newSpec);     // Update the system information database
            agentList.add(agentSpec.getId());
            // We keep an independent cached list containing the AIDs known by the queen
        }
}

if ( evt.getSource() == removeAgent ) {
        // Remove selected agent from the list and from internal list as well
        AID id = selectedAgentAID();
        System.out.println("selectedAgentAID is "+id);
        System.out.println("Corresponding node is "+findNode(id));
        if(id == null) {
            System.err.println("Cannot remove this. You may only remove agents from this list.");
            return;
        }
        // Remove the agent from the system information database
        OCLPAS.removeAgentsSpecification( (Q_UpdateMessage)
          OCLPAS.getAgentsSpecification().get(findOCLPASindex(OCLPAS,  id) ) );

        // Remove its link from the tree
        tree.getDTModel().removeNodeFromParent(findNode(id));
        // And remove it from the cached agents AID list
        agentList.remove(agentList.indexOf(id));
    }

    // Either RUN or STEP were pressed: Q_STEP has to be sent to all agents
    // whereas Q_RUN only to those selected
    if ( (evt.getSource() == stepButton) || (evt.getSource() == runButton) ) {
            System.out.println("Sending a STEP/RUN command to all agents");
            ACLMessage stepMsg;
            for (int i = 0; i < agentList.size(); i++){
              if (evt.getSource() == stepButton)
                    stepMsg = OCLPAgentTools.wrapMessage(myAgent,
                                                                    (AID) agentList.get(i),
                                                                    (new Q_STEP()),
                                                                    ACLMessage.PROPOSE,
                                                                    InteractionProtocol.FIPA_PROPOSE,
                                                                    ontology.getName(),
language.getName());
              else
                    stepMsg = OCLPAgentTools.wrapMessage(myAgent,
                                                                    (AID) agentList.get(i),
                                                                    (new Q_RUN()),
                                                                    ACLMessage.PROPOSE,
                                                                    InteractionProtocol.FIPA_PROPOSE,
                                                                    ontology.getName(),
language.getName());
              // Spawn a REInitiator behaviour
              addBehaviour(new SendStepBehaviour(myAgent, stepMsg));
            }
    }

    if ( evt.getSource() == jButton45 ) {                   // Upload OCLP to agent
            System.out.println("Attempting to parse and update OCLP");
            if ((receiver = selectedAgentAID()) == null)
              System.err.println("Error attempting to send OCLP program, no agent selected.");
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            Q_UpdateMessage updateMsg = new Q_UpdateMessage();
            OCLP_program newOCLP = new OCLP_program();
            if ((newOCLP = OCLPAgentTools.OCLP_program_fromString(text_OCLP.getText())) == null) {
              // text_OCLP is the text editor in the GUI

              System.out.println("An error has occurred trying to parse the OCLP."+
                                        " Please check syntax or report the error.");
            }else{
              System.out.println("\n OCLP parsed and is \n\n"+
                                        OCLPAgentTools.OCLP_program_toString(newOCLP));

              updateMsg.setNewOCLP(newOCLP);
              updateMsg.setDoOCLP(true);
              updateMsg.setNewSemantics(credulousSemantics.isSelected());
              updateMsg.setDoSemantics(true);

              updateMsg.setToAgent(receiver);
              updateMsg.setNickname(produceAgentName(receiver));
              OCLPAS.removeAgentsSpecification( (Q_UpdateMessage)
          OCLPAS.getAgentsSpecification().get(findOCLPASindex(OCLPAS, receiver)) );
              OCLPAS.addAgentsSpecification(updateMsg);

              myAgent.sendUpdateMessage(updateMsg);
            } // end method: actionperformed

    }

    if ( evt.getSource() == saveButton ) {                  // Save project button
            JFileChooser fileChooser = new JFileChooser();
            fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
            fileChooser.setDialogTitle("Save your OCLPAS project...");
            if (fileChooser.showSaveDialog(this.getParent()) != JFileChooser.APPROVE_OPTION)
              return;
            try{
              File file = fileChooser.getSelectedFile();
```

```
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(OCLPAS);
            oos.close();
         }catch(Exception exce){ return; }
   }

   if ( evt.getSource() == loadButton ) {          // load project button
         JFileChooser fileChooser = new JFileChooser();
         fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
         fileChooser.setDialogTitle("Save your OCLPAS project...");
         if (fileChooser.showOpenDialog(this.getParent()) != JFileChooser.APPROVE_OPTION)
            return;
         try{
            File file = fileChooser.getSelectedFile();
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            OCLPAS = (OCLPAS_system) ois.readObject();
            boolean doSpawn; // whether to spawn new agents or not
            String octpath = "./oct";          // optional octpath - user input
            doSpawn = askWhetherToSpawn();
            if (doSpawn)
               octpath = ((String)JOptionPane.showInputDialog(  Q,
                                                "Please enter OCT commandline: ",
                                                "Spawning new agents...",
                                                JOptionPane.PLAIN_MESSAGE,
                                                null,
                                                null,
                                                "./oct") );
            ois.close();
            agentList.clear();      // clear cached AID list
            Q.tree.getRootNode().removeAllChildren();    // clear GUI tree
            for (i = 0; i < OCLPAS.getAgentsSpecification().size(); i++) {
               newSpec = (Q_UpdateMessage) OCLPAS.getAgentsSpecification().get(i);
               // create a new agent
               if (doSpawn) {
                  receiver = createNewLocalOCLPAgent(newSpec.getNickname(),
                                                myAgent.getAID(),
                                                1234,
                                                octpath);
               }else{
                  receiver = newSpec.getToAgent();
               }
               // send UpdateMessage to initialise the fields in the agent
               myAgent.sendUpdateMessage(newSpec);
               // fill in cached AID list
               agentList.add(receiver);
               // add a link to the GUI tree
               agentSpec = new AgentSpec2();
               agentSpec.setId(receiver);
               agentSpec.setNickname(newSpec.getNickname());
               agentSpec.setChannelInput(newSpec.getNewCommChannels().getChannelInput());
               agentSpec.setChannelOutput(newSpec.getNewCommChannels().getChannelOutput());
               Q.tree.getRootNode().add(new DefaultMutableTreeNode(agentSpec));
            }
         }catch(Exception exce){ return; }
   }

}

public boolean askWhetherToSpawn() {
   // Asks the user, while loading a project, whether to spawn new agents or to use existent ones
   Object[] options = {"Spawn new agents in this platform",
                       "Use existing agents with the same names",
                       "Use existing agents and let me choose their new names"};
   String agentslist = "";
   for (i = 0; i < OCLPAS.getAgentsSpecification().size(); i++)
         agentslist += ((Q_UpdateMessage)OCLPAS.getAgentsSpecification().get(i)).getNickname() +
            "\n";
   int n = JOptionPane.showOptionDialog(Q,
                        "The following agents were found in this file: \n"+agentslist+"\n"+
                        "TOADs can create new agents in the Queens platform or let you choose"+
                        " from existing running agents (which will be overwritten)?",
                              "Loading a proect",
                              JOptionPane.YES_NO_CANCEL_OPTION,
                              JOptionPane.QUESTION_MESSAGE,
                              null,
                              options,
                              options[0]);

   if (n == 0) return true;
   if (n == 1) return false;
   int i; String newName; AID newID;
   for (i = 0; i < OCLPAS.getAgentsSpecification().size(); i++) {
         newSpec = (Q_UpdateMessage) OCLPAS.getAgentsSpecification().get(i);
         newName = ((String)JOptionPane.showInputDialog(Q,
                        "Please select the name of the living agent"+
                              " to update with "+newSpec.getNickname()+
                                    "'s data:\n",
                        "Agent "+newSpec.getNickname(),
                        JOptionPane.PLAIN_MESSAGE,
                        null,
         null,
         "") );
         newID = new AID( newName, AID.ISGUID );
         newSpec.setNickname( newID.getLocalName() );
         newSpec.setToAgent( newID );
   }
   return false;
}

};

}
```

Appendix C

OCLPAgentTools services

This is a list of the service functions privided in the OCLPAgent class.

- `public static String AnswerSet_toString(AnswerSet as)`
  // Returns a formatted String for the given AnswerSet object.

- `public static String AnswerSets_toString(AnswerSets as)`
  // Returns a formatted String for the given AnswerSets object.

- `public static OCLP_program copyOCLP_program(OCLP_program original)`
  // Creates a deep-copy of the given OCLP_program object.

- `public static OCLP_component copyOCLP_component(OCLP_component original)`
  // Creates a deep-copy of the given OCLP_component object.

- `public static OCLP_rule copyOCLP_rule(OCLP_rule original)`
  // Creates a deep-copy of the given OCLP_rule object.

- `public static OCLP_program upDateOCLP(OCLP_program oldProgram, AnswerSet AtomSet, String identity)`
  // Updates the given OCLP_program to include a new component under the name of "toads"+identity. The new component will be less preferred than all the others and include an assertive rule for each positive atom in AtomSet and a constraint rule for every negative atom in AtomSet.

- `public static String OCLP_program_toString(OCLP_program brain)`
  // Returns an OCT-compatible formatted String for the given OCLP_program object.

- `public static boolean AnswerSets_compare(AnswerSets ref_list1, AnswerSets ref_list2)`
  // Compares the given answer sets to see if they are essentially equal.

- `public static boolean AnswerSet_compare(AnswerSet ref_set1, AnswerSet ref_set2)`
  // Compares the two answer set objects to see if they are essentially equal. E.g. discarding repeated atoms.
- `public static OCLP_program upDateOCLP(OCLP_program oldProgram, AnswerSet AtomSet, String identity)`
  // update the given OCLP_program using the given set of atoms and name the new component "toas"+identity
- `public static OCLP_program OCLP_program_fromString(String program)`
  // An OCT format compatible OCLP program parser
- `public static boolean AnswerSets_compare(AnswerSets ref_list1, AnswerSets ref_list2)`
  // Compare the given AnswerSets to see if they are essentially the same
- `public static AnswerSets solveOCLP(OCLP_program program, boolean credulous, String OCT_PATH, String parameters)`
  // Invoke OCT to solve the given OCLP_program, with given semantics (credulous true for credulous, sceptic otherwise), oct path and additional parameters for OCT